TU WIEN Informatics

Smart Contracts in a DAG Ledger

Blockchain 5.0

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Lukas Hetzenecker, BSc
Registration Number 01225963

to the Faculty of Informatics

at the TU Wien

Advisor: Ass.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn Monika di Angelo

Vienna, 24th January, 2020

Lukas Hetzenecker	Monika di Angelo

Acknowledgements

My parents, Wilhelm and Hilda, for supporting me during my studies.

My advisors for this thesis, Monika di Angelo and Gernot Salzer, with whom I created the first “smart contracts” lecture at this university, that already provided me with a vast knowledge for cryptotechnologies beforehand.

Our seminar group for providing valuable input and inspiring discussions.

Susanne and Alice for giving feedback on the first draft of this thesis.

David Sønstebø, Dominik Schiener, Serguei Popov.

Vitalik Buterin, Gavin Wood.

Guido van Rossum, and the Python Software Foundation.

Linus Torvalds.

Abstract

Cryptocurrencies generated a lot of hype in the recent years. Following this hype lots of different cryptocurrency projects were started, each of them promising awesome novel features. We look at what those offer from a pure technical perspective. From our experience, it seems that smart contract platforms are getting especially popular, and rightly so – they open up a vast amount of new possibilities how users can interact with cryptocurrencies. We will look into how different platforms try to advance this field.

A current trend in blockchain research is coming up with solutions for improving the at this time still limited transaction throughput of blockchains, which arguably prevents more widespread use. We will look into proposals for increased scalability.

One of the most promising solutions for better scalability is challenging the entire structure of organizing transactions in a *blockchain*. This bottleneck can be replaced with a directed acyclic graph, which has better properties for transaction throughput. Although this comes with the disadvantage that smart contract engines are more difficult to implement on such ledger structures.

In this thesis, we researched how this apparent contradiction can be resolved to combine "Blockchain 2.0" technologies, that are smart contracts, and "Blockchain 3.0" technologies, how DAG ledgers are sometimes called, to bring the best of both worlds together. Such a unification could then fittingly be termed "Blockchain 5.0".

Contents

CHAPTER 1

Introduction

1.1 Blockchain

Blockchain — an open, distributed ledger made of blocks linked using cryptographic hash functions, with Bitcoin as its most prominent implementation — gained a lot of media attention in the recent years. In this context, distributed means that consensus is generated and maintained in a decentralized manner, and all members of the network can check and verify transactions. Two prominent designs for maintaining consensus are PoW and PoS.[13]

To give just one example, the research and advisory firm *Gartner* identified it as one of the Top 10 technology trends for 2019[26]. Whereas the scientific community largely ignored Bitcoin when it was first introduced, and only took it seriously after practice showed it worked ([55]), subsequent developments often have an active research community available from the start of the project.

Blockchain can be seen as a decentralized database, containing a public ledger of transactions that were executed between participants of the decentralized network. Every transaction gets verified independently by every participant, and a consensus is formed by the majority. A transaction creates an irrefutable record in the public ledger. This method frees the participants from the need of any centralized trustworthy arbitrator or third party — one of the main reasons why Satoshi Nakamoto published the Bitcoin protocol in January 2009, following the financial crisis.

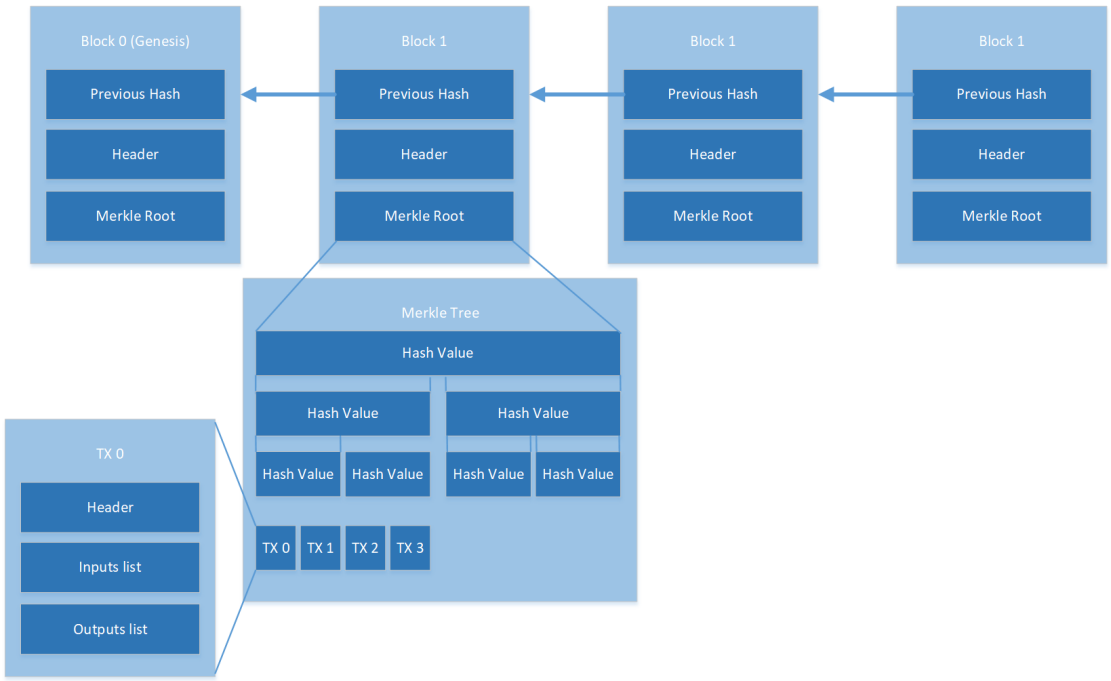

Figure 1.1: Blockchain 1.0 structure overview

Figure 1.1 shows the structure of a blockchain, with its name-giving structure of blocks, back-linked until the very first *Genesis* block, at the top left in this figure. Every block contains an ordered list of transactions, whose fields are hashed, and these hashes are again structured in a *Merkle Tree*. The root of the tree, the *Merkle Root*, is the only field needed in the header of the blocks. This simple, but powerful, structure forms the fundamental forgery detection capability of blockchains — because any change to any transaction carried out in retrospect would result in a change of every single succeeding hash. Although used in cryptocurrencies, this feature is much older than that: cryptographically secured chain of blocks date back to 1991, when they were described by Stuart Haber and W. Scott Stornetta.[13]

Currently the most active research topics include the limited scalability of blockchains[86]: estimates put transaction processing capacity between 3.3 and 7 transactions per second for the Bitcoin blockchain[18], which severely limits its applicability as day-to-day payment coin.

1.2 Blockchain 2.0 - Smart Contracts

While the goal of Bitcoin was to provide a form of electronic cash, the next evolutionary step of blockchain technologies allowed autonomously executing algorithms without the need of any third party or middleman. In literature such platforms were often called "Blockchain 2.0" (e.g. [43]). They are the foundation for decentralized applications, or in

short *DApps*[3]. Several lectures at the TU Wien delve into the development of DApps on the *Ethereum* platform, the first that enabled the so-called *Smart contracts* in a cryptocurrency.

But the concept of smart contracts predates Ethereum's release in 2013, Nick Szabo mentioned digital contracts that form a set of promises already back in 1996[78]:

> New institutions, and new ways to formalize the relationships that make up these institutions, are now made possible by the digital revolution. I call these new contracts "smart", because they are far more functional than their inanimate paper-based ancestors. No use of artificial intelligence is implied. A smart contract is a set of promises, specified in digital form, including protocols within which the parties perform on these promises.
>
> – Nick Szabo, 1996

Szabo mentions four key principles of contracts, derived from common law: observability (that principals can observe the others performance, or prove their own), verifiability (the ability of a principal to prove to an arbitrator that a contract has been performed or breached), privity (knowledge and control over a contract should be distributed among parties only as much as is necessary for the performance of that contract) and enforceability.[79]

Bartoletti et al.[6] defined them as "computer programs that can be consistently executed by a network of mutually distrusting nodes, without the arbitration of a trusted authority".

Trüeb[80] noted that smart contracts lack a clear-cut definition. They can be many different things: (i) an automation process; (ii) a software script or program; and (iii) the means by which blockchains or alternative ledger technologies will finally come into the mainstream.

But while the term *smart contracts* would suggest that they are contracts in some legal sense, this is not the case, as smart contracts lack the elements of offer, acceptance and consideration typically found in legal contracts. Now that the term smart contracts has already become a quasi-standard for such decentralized programs, Vitalik Buterin, the founder of Ethereum, regrets choosing it for his implementation in Ethereum[1]:

> To be clear, at this point I quite regret adopting the term "smart contracts". I should have called them something more boring and technical, perhaps something like "persistent scripts".
>
> – Vitalik Buterin, 2018

[1] https://twitter.com/VitalikButerin/status/1051160932699770882

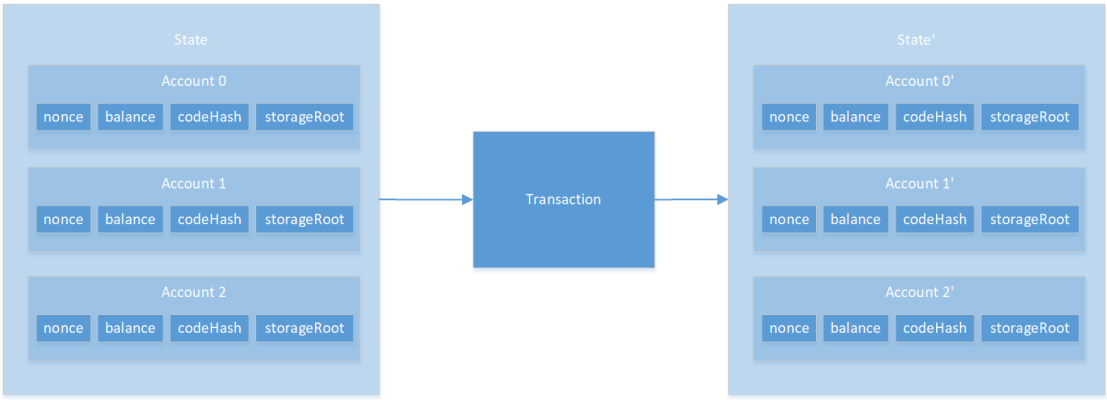

Figure 1.2: Blockchain 2.0 (Smart Contracts) structure overview

Ethereum can be seen as a state machine, whose *contracts* encode the state transition functions.[8] Only the valid updates to the contract states are recorded on the blockchain, to ensure their correct execution.[6]

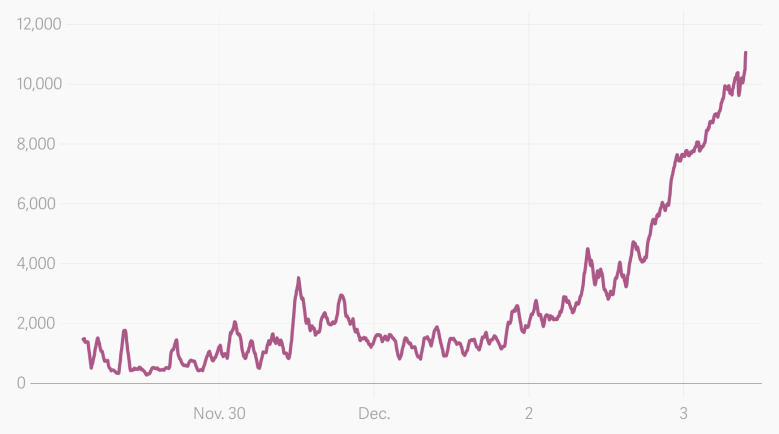

Figure 1.3: Pending transactions in the Ethereum network after the release of *CryptoKitties*[85]

While the concept of smart contracts seems revolutionary, at least in the space of blockchain technologies, they cannot solve the problems with scalability. In fact, smart contracts tend to make this problem even worse, as the probably most prominent example for this, the DApp *CryptoKitties*, shows. This app, the first big online game on Ethereum, enables players to buy, trade and breed digital cats. When it got really popular in December 2017 it caused a congestion in the Ethereum network within one week of its launch, leaving many transactions pending (figure 1.3) and resulting in a huge increase of transaction fees. During its most popular times, it accounted for more than 13% of all transaction traffic in the Ethereum network. As solution to this growing problem, the developers of the app increased the costs to interact with the app, particu-

larly the birthing fee of the kitties. This reduced the number of people willing to breed these digital kittens, and therefore the traffic in the network.[36]

This example highlights that Ethereum suffers from the same fundamental scalability problem as other blockchains like *Bitcoin*, presently only managing a global transaction rate of 15 transactions per second. An often talked about possible solution to this is *Sharding*, which will split the blockchain into discrete parts. While originally planned to activate with Ethereum's *Casper* upgrade, this feature got postponed to some later date[15]. As of January 2020, no reasonable roadmap has been published yet.

That being said, the public interest in blockchains and smart contracts is still unabated, as the search trend in Google (figure 1.4) shows.

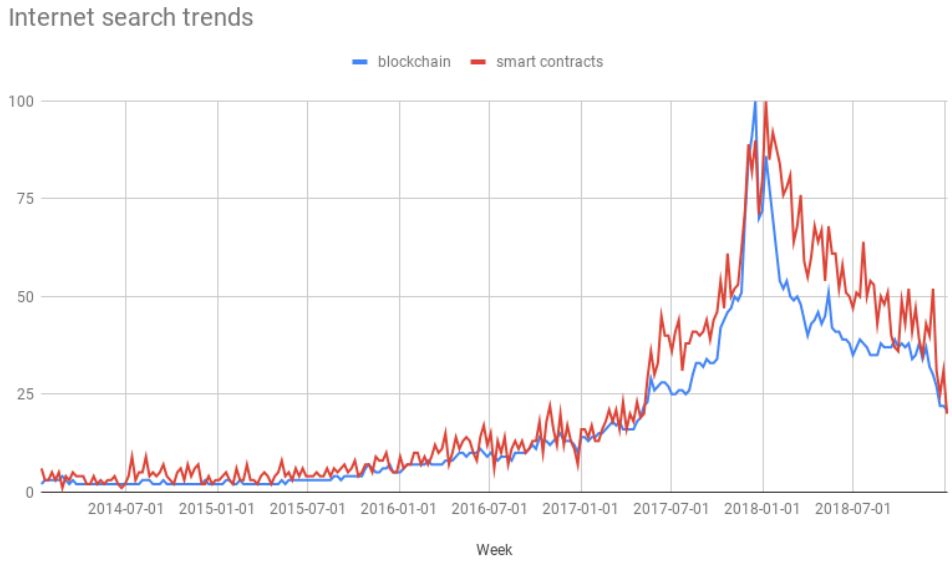

Figure 1.4: Popularity of search terms *blockchain* and *smart contracts*, source: Google Trends

1.3 Blockchain 3.0 - Directed acyclic graphs

An even newer generation of cryptotechnologies, sometimes called "Blockchain 3.0"[59], tries to overcome these problems. They are still named blockchain, although that is actually a misnomer, because many such implementations no longer order transactions in a linked list. Cryptocurrencies such as *IOTA*[65] or *NANO*[54] use a DAG instead of a chain to store its ledger. The big advantage hoped to be achieved with this method is getting rid of the inherent transaction rate limit, therefore accomplishing unlimited scalability (at least in theory). IOTA also does not use the concept of mining, resulting in feeless transactions for its users.[3].

Although this iterations of Blockchain provides desirable properties, combining the smart contract capabilities of Blockchain 2.0 with the DAG ledger structure of Blockchain 3.0 is still an unsolved problem. Due to the graph structure of e.g. IOTA, no total ordering of all transactions is possible - a requirement for existing smart contract platforms.

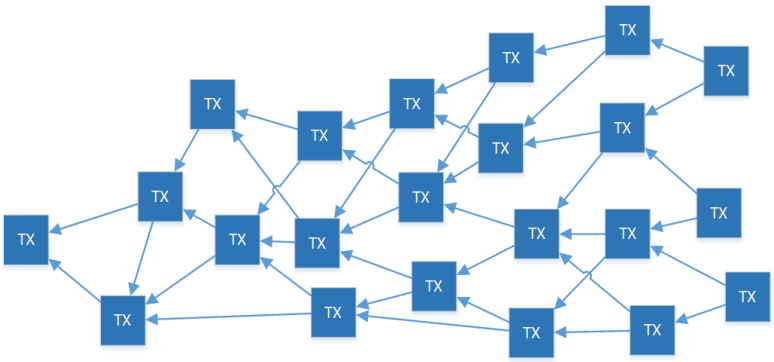

Figure 1.5: Blockchain 3.0 (DAG) structure overview

A blockchain consists of, as the name suggests, blocks that are linked. Blocks contain possibly zero or more transactions, and one block must strictly follow the previous one. This is done by adding a hash pointer to the previous block into the block header.

In mathematics, such a construct is called a *chain* (or a *totally ordered set*). This set is paired with a total order, a binary relation ($\leq$) on a set X, that is reflexive, antisymmetric, transitive and total.[69] Formally these properties are defined as follows: A binary relation $\leq$ is a total order on a set X if the following statements hold $\forall x, y, z \in X$:

1. Reflexivity: $x \leq x$
2. Antisymmetry: $x \leq y \wedge y \leq x \implies x = y$
3. Transitivity: $x \leq y \wedge y \leq z \implies x \leq z$
4. Totality: $x \leq y \vee y \leq x$

Examples for such totally ordered sets, apart from the set of real numbers ($\mathbb{R}$) ordered by the usual less or equal than relation ($\leq$), are the transactions in a blockchain. Miners are responsible for this order by establishing links to predecessor blocks, and the order of transactions within a block (which doesn't necessarily mean that was the order in which the transactions were made).

A DAG on the other hand is a more relaxed data structure. Similar to a blockchain it contains a finite amount of vertices (blocks), but here the blocks are linked in a directed graph, instead of a chain. With this the *Totality* axiom is lost, as some elements can be

incomparable. The *Reflexivity*, *Antisymmetry*, and *Transitivity* axioms are still fulfilled though. Such sets are called *partially ordered sets* (or *posets* for short).

This definition shows that every blockchain is also a DAG (with only one outgoing edge per block), but the reverse is not true.

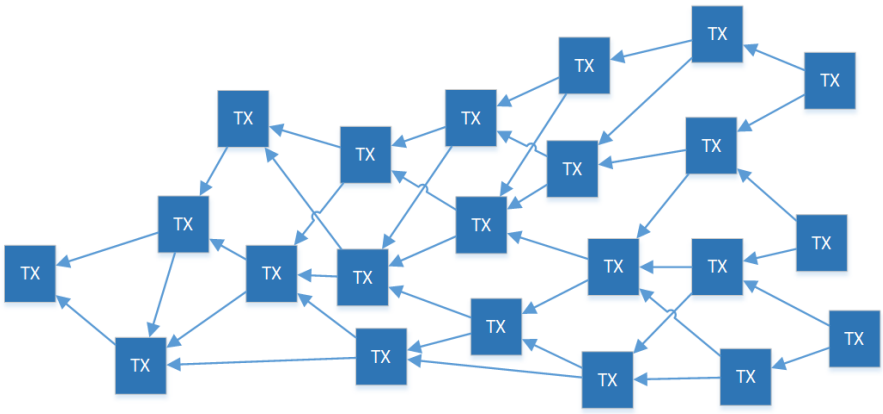

Figure 1.6: Structure of the DAG in IOTA (Tangle)

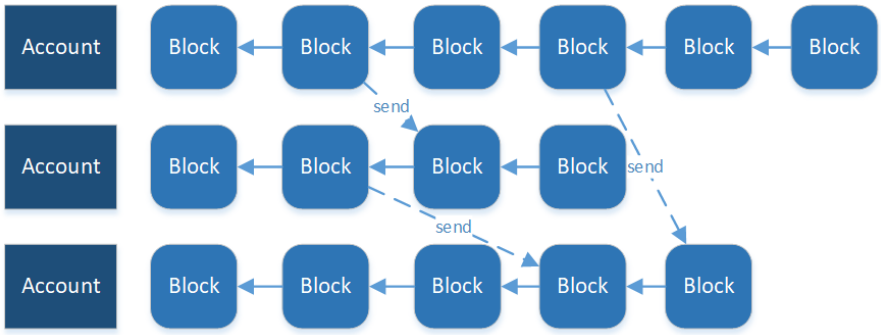

Figure 1.7: Structure of the DAG in NANO (Block lattice)

Figures 1.6 and 1.7 show different implementations in distributed ledger technologies. Both of them fulfill the properties of a DAG.

1.4 Research Questions and Methodological Approach

The analysis of those mentioned problems should lead to the answer of the following Research Questions:

1.4.1 RQ1 - What do current smart contract platforms have in common?

The purpose of this research question is to analyze currently existing smart contract platforms, to find what they have in common and where they diverge from one another. We are interested in the implementations of the currently most popular cryptocurrencies (as ranked by coinmarketcap.com) that support some form of smart contracts, in particular which execution engines (virtual machines or VMs) they are based on, and how developers can program them.

1.4.2 RQ2 - How do smart contract VMs work in detail?

The main focus here will be on Ethereum and 1) how transactions are executed by the EVM, and 2) how the order of transactions is established. Race condition attacks are unavoidable in Ethereum, like the *Transaction-Ordering Attack*, where a miner can choose the order in which the transactions are mined in a single block.[16]

1.4.3 RQ3 - How can the basic requirements of smart contracts be applied in a DAG?

As already discussed, is is generally impossible to determine the order of transactions in a DAG. The difficulty lies in the lack of total order of the transactions. As there is only a partial order, there does not need to be a directed edge between two transactions, A and B, and therefore one cannot say in general whether A or B happened first. This makes it hard (or generally impossible) to implement smart contracts in such a structure. To solve this, simplifications on the data structure of the transactions have to be made. Besides introducing timestamps to the tangle, as Serguei Popov proposes[64], other solutions without timestamps will be discussed as well.

1.4.4 RQ4 - How can a consensus of asynchronous transactions be formed in a DAG with uncertain timestamps?

While the focus of RQ3 was on data structures in a DAG, RQ4 will deal with consensus mechanisms. One of the most important feature of cryptocurrencies is that consensus is maintained in a decentralized manner, therefore freeing everyone from the need of a centralized trustworthy arbitrator or third party[13]. RQ4 will analyze consensus algorithms, that can work with smart contracts in a DAG ledger.

1.5 Related literature

Trüeb [80] provides approaches to classifying smart contracts from a legal perspective. He notes that this can be seen from different angels, ranging from being not a contract at all, but just an automated closing equipment, to being a novation of human consent

in binary data and therefore a contract in its own right. A rough high-level overview of features of smart contracts is provided as well, giving examples of application fields.

Bartoletti et al.[6] empirically documents how smart contracts are interpreted and programmed on various blockchain platforms.

Jiménez[63] analyzed in his masters thesis the state of the Ethereum blockchain. It contains a survey about the inner working of the platforms, going into the details of the database, application ecosystem and libraries. Furthermore a useful tool [2], written in Python, is provided to query the state and generate statistics of the Ethereum blockchain.

1.6 Aim of the Work

First, we analyze platforms for smart contracts in regard to underlying concepts, status, and issues. We compare different platforms to support developers in choosing a suitable platform for their DApp.

Next, we want to combine the two promising research topics — smart contracts and DAGs. Based on our platform comparison, we investigate how a smart contract engine could use a DAG data structure - and how this does compare to existing concepts like sidechains.

At the moment, there is no known implementation of smart contracts in a DAG. This thesis should act as a guide to the various kinds of smart contracts possible in a DAG and to the impossible ones — due to the graph structure of the ledger.

The classification of smart contracts that can or cannot be implemented on a DAG is currently of particular interest to the IOTA foundation and this thesis should advance the research in that area.[48]

Another goal is to develop a simple prototype, which applies the previously researched concepts.

1.7 Methodological Approach

1.7.1 Survey of platforms

Out of the top 50 cryptocurrency projects with the highest market capitalization (as found on `coinmarketcap.com`), we analyze whether their platforms offer any support for persistent scripts, or any other form of smart contracts. We check how powerful those are, and only consider *Turing complete* platforms for a more in depth review. We then examine their execution environments to find out how they can be programmed, and try to find similarities between them.

Another crucial criteria for developers is the number of users they can reach with their DApp, so we try find sources for the activity in their underlying blockchains as well.

[2] `https://github.com/carlesperezj/ethereum-analysis-tool`

1.7.2 Prototype

A goal of this thesis is to develop a basic prototype application, that showcases the implementation of smart contracts, or more precisely of a smart contract virtual machine, in a DAG ledger. We show how this virtual machine can be adjusted and extended, to better deal with the peculiarities of such DAG ledger structures.

1.7.3 Scope & Limitations

We limit ourselves to a literature study to get an overview of the smart contract and DAG space, and do not perform a deep technical analysis into every platform. We only pick a representative technology from each of these fields — Ethereum and IOTA — where a more in-depth review into certain technical aspects is conducted, as they are the basis for the subsequent prototype application.

The prototype is a minimum viable product that embeds a subset of smart contract features into a DAG. It lacks certain key aspects like dealing with value transfers of a native currency by updating the balance of an account. The final structure will be a block lattice, similar to sidechains, but we will not implement interactions between different chains in the DAG.

As the focus is on the block structure, the prototype has only a very simplified model consensus, where a central node is the only authority and responsible for achieving consensus by periodically issuing blocks, which are the only source of truth in this chain. Classical methods for consensus like *Proof of Work* or *Proof of Stake* could be applied to this chain, but are out of scope for this work.

1.8 Structure of the Work

The rest of this thesis is organized as follows:

At the beginning in chapter 2 the survey of existing smart contract platforms will be performed. We are mostly interested in their inner workings, how their smart contract engine operate, and how contracts can be programmed on them. We try to describe what sets the platforms apart from each other. This should lead to a general understanding on the space of smart contract platforms, and what is required to implement such engines. We also want to know which platforms in this field are the top dogs, that *DApp* developers need to understand.

After the basics of *Ethereum* are explained in chapter 3, we will dip into the inner workings of its *virtual machine* by looking at how exactly transactions are broadcasted and executed.

In chapter 4 the DAG ledger *IOTA* with all its quirks is analyzed, again with focus on the publishing of transactions, and on a method for achieving consensus in such a network.

Chapter 5 delivers the theoretical groundwork on how smart contracts calls can be embedded into DAG ledgers, combining the lessons we learned in all previous chapters.

In chapter 6 we will then bring the previous theoretical work to fruition, by introducing a simplified *Proof of Concept*, that is able to embed smart contract calls into IOTA's DAG ledger. But first, a smart contract engine has to be chosen and extended, and ways to generate message streams in IOTA have to be found.

Finally chapter 7 will summarize the findings, and highlight potential future work, that was out of scope for this thesis.

CHAPTER 2

Platforms

For this chapter, a literature study about smart contract concepts and existing platforms was performed. The main focus was on currently existing platforms that are in a useable state and how they differentiate themselves. Of particular interest were novel approaches and how they try to overcome scalability issues.

Overtorment[61] had already compiled a curated list of platforms to run smart contracts, which was used as starting point for our research.

Bartoletti et al.[6] already did a similar survey in 2017, where they highlighted the key differences between the most popular platforms at that time. They used the amount of articles on the `coindesk.com` portal as metric for the popularity.

2.1 Most popular smart contract platforms

Because smart contracts are seen as the next evolutionary step of Blockchain technologies, there are a variety of different platforms with diverse objectives that support some form of a smart contract engine. This chapter establishes a common foundation for such platforms. The result is a listing of properties those platforms have, although the details of the implementation of course diverge between the examined platforms.

To research platform properties, a list of currently existing platforms is required. As the cryptocurrency space is evolving daily, new platforms are developed rapidly, and no relevant scientific literature could be found that is still up to date. Papers older than six months are generally already too old to provide satisfactory information. While we try to provide accurate, up-to-date information, we acknowledge that our work will share the same fate.

Based on the definition of smart contracts in chapter 1, we assemble a list of the currently most popular platforms. We use the existence of an Turing-complete language as basic

requirement. The ledger structure needs to follow the definition of cryptocurrencies, which includes a decentralized consensus (consensus that is generated and maintained in a decentralized manner, where agents can check and verify transactions).

Furthermore, we restrict the acceptable platforms to the following minimal constraints that Bartoletti et al.[6] specified as well: the platform needs to (i) be already launched, (ii) be running and supported from a community of developers, and (iii) be publicly accessible.

We limited our research to the Top 50 cryptocurrencies by market capitalization from the list provided by coinmarketcap.com. The research results can be found in Appendix A.

2.2 Ethereum VM

EVM, the virtual machine used by *Ethereum* and several other projects, is a "quasi turing-complete" (the available gas for a transaction limits the instructions that can be executed[30]) 256-bit stack-based execution environment for smart contract bytecode.

Haifeng[30]noted several flaws in the design of the EVM, that can lead to security concerns. The *lack of a standard library* makes otherwise easy functionality like string splicing, cutting and searching difficult to implement, and forces developers to borrow such code snippets from other projects, which probably lack the proper auditing. *DApps are difficult to debug and test*, often throwing *OutOfGas* exceptions without reporting useful error informations. *Floating point numbers are not supported*, and *Contracts cannot be upgraded*, hindering the ability to implement security patches for existing deployments. We are also aware of these issues and share this assessment, but want to add that projects like *OpenZeppelin Contracts*[1] want to become the de-facto standard library, recent versions of *Solidity* added the support for returning human-readable error messages[2], and pattern for upgradable contracts do exist[3], which at least shows that the community is effectively working on decreasing these teething problems.

Another shortcoming is the word length of 256 bits, which differs from the usual 64 bit word length of modern CPUs, adding unnecessary overhead, wasting storage and decreasing the computational efficiency[32]. This word size was chosen to easily place *Keccak256* hashes into a single word.[60]

Nevertheless Ethereum still seems to be the most popular and widely used platform today, and despite the fact that several newer ones emerged, none of them could dethrone Ethereum from this spot.

The "DappRadar 2019 dapp Industry Review"[42] also shares this assessment, stating that "Ethereum remains the most significant smart contract blockchain. It grew the

[1] https://openzeppelin.com/contracts/

[2] https://github.com/ethereum/solidity/releases/tag/v0.4.22

[3] https://kauri.io/how-to-write-upgradeable-smart-contracts-with-truffle-5.0-and-zeppelinos-2.0/315cbd6c71574e2686e15f0a20003089/a

daily user base of its dapp ecosystem 118% in 2019, with daily value up 166%. None other emerging smart contract blockchain demonstrated a sustained audience of dapp usage in 2019."

Because the EVM is such a popular and well-established smart contract engine, it is also utilized in several other cryptocurrency platforms. A small subset of those include:

Ethereum Classic is a direct fork of the Ethereum chain in response to the *DAO hack*[4] of June 2016, in which hackers could drain the funds of the DAO, a smart contract on the chain. This caused frictions in the Ethereum community, as the Ethereum developers decided per a vote to recover the stolen funds by putting them into a new smart contract and allowing the original owners to restore their funds. Community members advocating for blockchain immutability, and the concept of "code is law"[5] rejected that reversal and kept the unforked version of Ethereum under the name *Ethereum Classic*.

This project is still active, but no big development seems to have happened for it. It has not implemented other feature forks of Ethereum[6], and therefore is still mostly on the feature level of the *Homestead* fork. It had a few forks on its own, but these have mostly focused on delaying the difficulty bomb explosion (see sec.3.2)[7], or adjusting the monetary policy[8]. The only feature update, *Atlantis*, adding more opcodes, precompiled contracts and zk-SNARKs activated in September 2019[9].

Another goal of the Ethereum Classic team is to develop an independent virtual machine implementation in *Rust* under the name *SputnikVM*, which unites all Ethereum'esque blockchains (Ethereum Classic, Ethereum, Ellaism, Ubiq, etc.)[10].

Tron developed a VM with full compatibility to the EVM, which will be described in greater detail in section 2.4.

Qtum, a Proof-of-Stake blockchain combining the UTXO (Unspent Transaction Outputs) model of Bitcoin with the EVM of Ethereum as smart contract machine[11].

VeChain is a public blockchain trying to "allow efficient and transparent transitions (upgrades) of the protocol to adapt to new challenges", and supporting a "native fee delegation"[12], while still being closely linked to the EVM. It still has the same account

[4] https://vessenes.com/deconstructing-thedao-attack-a-brief-code-tour/
[5] https://www.vice.com/en_us/article/z43qb4/the-ethereum-hard-fork-spawned-a-shaky-rebellion-ethereum-classic-etc-eth
[6] https://github.com/etclabscore/go-ethereum/blob/development/core/config/mainnet.json
[7] https://github.com/etclabscore/ECIPs/blob/master/ECIPs/ecip-1010.md
[8] https://github.com/etclabscore/ECIPs/blob/master/ECIPs/ecip-1017.md
[9] https://www.coindesk.com/ethereum-classic-successfully-forks-improving-interoperability-with-ethereum
[10] https://github.com/ETCDEVTeam/sputnikvm
[11] https://en.bitcoin.it/wiki/QTUM
[12] https://doc.vechainworld.io/docs/overview

model, EVM, modified Patricia tree, and the RLP encoding method of Ethereum. For consensus, a modified variant of the *DPoS* method is used.[13].

2.3 EOS

EOS is one of the strong contenders to *Ethereum*, providing its own virtual machine and smart contract platform. The main thing that sets the two apart is the *DPoS*[14] consensus algorithm, reducing the requirements for miners and computation power, and therefore allowing transactions without fees. Instead users have to stake a certain amount of *EOS* tokens to utilize computing power, storage capacity or bandwidth.[73][21]

The *EOS VM*[15] is a WebAssembly engine based on the *LLVM* compiler project, and designed for fast (to achieve the high transaction throughput of EOS) and deterministic execution (non-deterministic rounding modes of floats in hardware are replaced by a deterministic software-based float arithmetic implementation).

Developers write the smart contracts in languages supported by *LLVM*, which currently are C, C++ and Rust. This can be seen as a barrier for developers, as those languages are perceived to be harder to learn, when compared to *Solidity* or *JavaScript*.

Although many sources still claim that its virtual machine will eventually be compatible with the EVM and its successor, the eWASM VM, and be able to execute EVM contracts with little adaptations in a sandboxed environment, as the original whitepaper stated, this feature got removed at a later revision of the whitepaper. No compatibility to EVM or eWASM is currently on the roadmap.[77]

2.4 TRON

The virtual machine of Tron, the *TVM*, was designed to be largely compatible with Ethereum's *EVM*. Smart contracts for Tron are also written in Solidity, and the only adaptation for existing source codes is replacing the hardcoded *ether* term with *sun*[16] (named after its founder).

This even goes so far that their technical standard for tokens, named TRC-20, is actually the same as the ERC-20 tokens (see 3.5.2 for more details of this standard) in Ethereum.[17]

[13] https://medium.com/@totientlabs/why-and-how-to-port-from-ethereum-to-vechain-e18e3f474848

[14] *Delegated* means that users can vote with their tokens to elect 21 (in the case of EOS) delegates, who will then produce the blocks

[15] https://github.com/EOSIO/eos-vm

[16] https://developers.tron.network/docs/converting-ethereum-contracts-to-tron

[17] At this point, it has to be mentioned that TRON also natively supports Token transfers via the TRC-10 standard

A reference of opcodes can be found in the documentation[18], which shows that they match those of the EVM, although lacking the ones introduced in later EVM upgrades like *Constantinople*.

For the consensus algorithm, this project has borrowed the pseudo-decentralized consensus of *EOS*.

2.5 Cardano

Cardano is the platform behind the *Ada* cryptocurrency with a strong academic background.

In the *Goguen* work scope, they have defined several projects with the goal to integrate smart contracts into their platform. This includes *Plutus*, a smart contract development platform and functional programming language based on *Haskell*. Another focus is on a simplified smart contract environment, focusing solely on financial application. But this milestone also encompasses another project, *KEVM*, which will be a formally-verified smart contract virtual machine compatible with the EVM. For a more secure implementation, it uses formal semantics for elements such as the configuration and transition rules of the EVM.

2.6 NeoVM

The NeoVM is the virtual machine for the cryptocurrency Neo. While not directly mentioned in the NEO whitepaper [58], a look into the source code[19] reveals that the NeoVM also acts as a state-machine, conceptually similar to the EVM.

Figure 2.1 shows the current architecture overview of the NeoVM. It consists of a *Execution engine*, which loads the bytecode, and pushes it alongside its related parameters into the *InvocationStack*, which stores execution contexts of different smart contracts, and the *ResultStack*, which stores the results after all scripts have been executed. The execution engine takes an instruction from current context, and then executes corresponding operations according to the instruction. The OpCodes contain similar arithmetical and logical instructions to the EVM, but also more instructions specifically for cryptographic operations, namely hashing via SHA-1, SHA-256, Hash160 (SHA-256 followed by RIPEMD-160), Hash256 (twice with SHA-256), and signature verification.

Neo currently supports smart contract development in the .NET languages C#, VB.Net and F#. For Java only a compiler with “basic features” is provided, and Python, Go, and JavaScript are planned to be supported sometime in the future.[5]

Another novel approach by Neo is the used consensus mechanism **Delegated Byzantine Fault Tolerant (dBFT)**, which provides high throughput and single block finality.

[18] https://developers.tron.network/docs/energy-costs-table
[19] https://github.com/neo-project/neo-vm

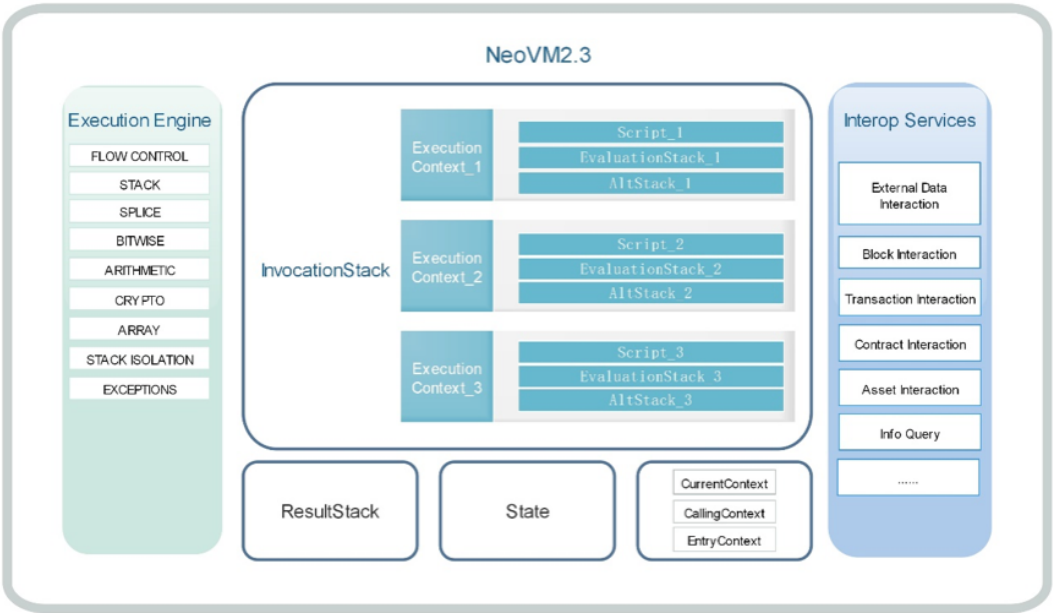

Figure 2.1: NeoVM architecture [57]

Marco Bareis wrote his diploma thesis about a "Comparison of Ethereum and NEO as smart contract platforms"[5]. This comprehensive work goes into all details that differentiate these two platforms. He argues that both platforms look similar on a superficial level, and because of their Turing-complete smart contract languages would theoretically also support all kind of smart contract applications, but that does not mean they share the same focus and goals. While Ethereum favors decentralization, but has no particular application domain, Neo wants to create a smart economy and therefore focuses on certain aspects like tokens or digital identity. Neo also does not seem to be as mature as Ethereum, as it even lacks key documentation papers, like a proper specification of its virtual machine. While Neo is currently theoretically superior in metrics like the throughput of transactions (although this should be taken with a grain of salt and the actual figures investigated, as there currently is no application on the Neo mainnet which requires a high throughput), Ethereum will eventually catch up as well with their new major release. Bareis further claims that, "if NEO cannot verify its claims regarding scalability, its role as globally used smart contract platform is dubious and developers need to ask themselves if the reduction of decentralization in NEOs consensus protocol is actually justified".

The "DappRadar 2019 dapp Industry Review"[42] also only found moderate activity on this platform, with currently 250 unique daily active wallets on average from 13 DApps that are being tracked, most of them in the *Games* category.

2.7 IOTA (Qubic)

Although this platform does not fulfill the previously established requirement of having an actual smart contract-capable product that is already launched, we briefly discuss it, as later chapters of this thesis need to reference here.

The DAG ledger IOTA is also currently developing their own solution, which is similar to smart contracts. It allows the execution of quorum-based computational tasks. They follow an event-driven architecure style, continuously listening for new matching input transactions on the Tangle, and starting the execution whenever their input changes. Their results get then again published into the Tangle, which in turn could cause the execution of other Qubics, which are listening to this output.[37]

For the development of these *Qubic* programs a new functional programming language (or more strictly, a dataflow programming language) called *Abra* was developed.

At the time of writing this thesis, only the whitepapers of *Qubic* and *Abra* are available, and not much else is known about these projects. Particularly no source code is yet published, so this conceptual platform can not yet be analyzed in detail.

The articles published so far suggest, that the programming language will not be Turing-complete (due to not supporting unbounded loops etc.), and that the platform in general will not be similar to e.g. Ethereum's smart contracts.

The developers explained their motivations and the limitations of *Qubic* in the following post:

> We've always said that Qubic won't be able to move funds directly and that the possibility to do that would be a layer on top through our Gateway concept. Most people, when they think of Smart Contracts, think of the ETH version of SCs. We're not building ETH-type SCs. We're building something completely different that in the end will most probably be able to do similar things. I think we first need to get clear on what Smart Contracts really are, because everyone seems to have a different view on those. [66]

2.8 Lisk

Lisk supports the execution of Turing-complete smart contracts, written in JavaScript (optionally as Node.js app). The determinism of executions is not ensured by the language itself like it is in Ethereum, but rather the developers need to make sure the applications are deterministic, e.g. by not using functions like *Math.random*.

Further information to Lisk can be found in the paper by Bartoletti et al.[6].

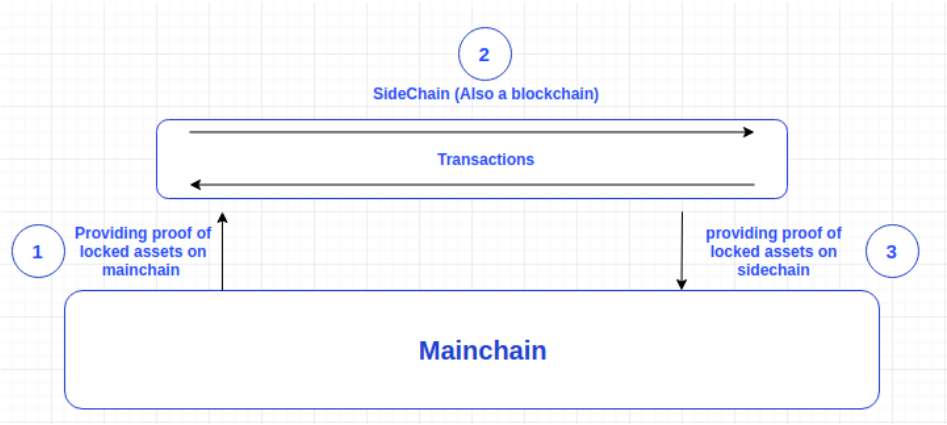

Figure 2.2: Sidechain [68]

2.9 Solving Scalability With Sidechains

The concept of pegged sidechains was already established in 2014 (shortly after the proposal for Ethereum in 2013) by Back et al.[4] They claimed that no big technical or economical innovations are possible in well-established blockchains like Bitcoin, because changes to the consensus mechanism have to be handled very conservatively. In this paper a method for transferring assets between different blockchains is described, that enables new blockchains with novel ideas that spur technical and economical innovation by allowing changes to critical parts, while still utilizing the assets they already own. They also note that such exchanges are possible via a method called *atomic swaps*.

Concepts recently proposed for solving scalability problems of Ethereum are often second layer solutions implemented via state channels. Those behave in the same way as payment channels (like the *Lightning Network* in Bitcoin), where all transactions are moved off-chain, and only uses the blockchain for settling the transactions when the channel gets closed. Participants interact directly with each other by exchanging state updates, which could be submitted at any time to the blockchain, but do not have to before the channel is closed. Only the participants themselves know of these intermediate state updates, because they are off-chain and do not get published to the whole network. This is in stark contrast to sidechains, which are complete permanent blockchains in their own right, and enable the interchangeability of assets via a two-way peg (see figure 2.2).[68]

There are many variants of this concept, ranging from tightly integrated implementations like *child chain* in the platform *Ardor*, where the child chains are also secured by the main chain validators (enabling ledger pruning and reducing the blockchain bloat, but not actually helping to improve throughput), to the *bridge chains* in *Ark*, that can operate completely independent and even employ different consensus mechanisms like *Proof of Stake* or *Proof of Work*.

Recent articles like “Solving scalability of Ethereum through Loom Sidechains” by Sar-

necký[71] argue, that increasing block sizes to process more transactions in the same time leads to increased centralization of the network as a whole, because fewer miners can participate in the progress due to increased energy requirements. They also argue that different smart contract applications have different security requirements, e.g. while smart contracts that transfer millions of dollars between wallets need the level of security Ethereum offers, not all transactions do. Use cases like simple data manipulations might get by with lesser security, as hackers have fewer incentive in exploiting such DApps. They introduce a concept, where the Ethereum blockchain acts as a core sidechain, but every dapp lives in its own sidechain. The communication between the chains happens through another second layer technology of Ethereum called *Plasma*.

This concept can also be seen with *Lisk*, which is "not a smart contract system", but rather a "custom blockchain system". Every single application lives in a completely separate, isolated sidechain, and is only responsible for itself. When a sidechain fails, the developer of that sidechain is to blame for it.[53]

Another example for a sidechain pegged to Bitcoin is *RSK*, a smart contract development platform. Although pegged to Bitcoin, their platform inherits key concepts from Ethereum, such as its account format, VM and web3 interface, and therefore *RSK'S VM* is backwards compatible to the *EVM*, meaning that compilers, tools, and DApps are compatible as well. The security of the platform is guaranteed via a *merged mining* process with Bitcoin, where miners can mine two cryptocurrencies that use the same algorithm together by embedding the block id from the secondary blockchain in the block of the primary blockchain.[47]

Also several suggestions for *Proof-of-Work Sidechains*[44] and *Proof-of-Stake Sidechains*[27] in the cryptocurrency *Cardano* have been published.

2.10 Conclusion

The following conclusion can be drawn from this investigation:

Many coins are simple forks. When changes to the validity of the rules for blockchain platforms are introduced, the blockchain splits into different chains. Using Bitcoin as an example, the following spinoffs were originally based on the same codebase: Bitcoin Cash, Litecoin, Bitcoin SV and Bitcoin Gold. Because of their shared origin, their features only diverge in certain details. For example, currently none of them support any Turing-complete smart contracts, they just have a simple scripting system. The exception here is Bitcoin Cash, which seems to have the programming language *Spedn* in development — but not many details about that can be found, and no code is yet publicly available.

Many smart contract platforms are still in development. The cryptocurrencies *XRP* and *IOTA* are currently in the progress of developing a smart contract platform, *Cardano* released their implementation in a testnet in August.

The Ethereum VM is the most popular platform and not only used by Ethereum, but also by other cryptocurrencies. It was Ethereum that introduced smart contracts, and due to their first mover advantage it is still the most popular platform. But surprisingly, not only Ethereum and the direct fork Ethereum classic use the EVM, but also other cryptocurrencies make use of it. Qtum, which is a fork of Bitcoin, is fully EVM-compatible. RSK develops a platform, that extends the EVM, but is still fully backwards-compatible to it. And even platforms that have their own smart contract engine provide at least a compatibility layer for the EVM (e.g. Cardano). The reason for this choices are that the EVM is already well-known and well-used.

There is not much variety in the smart contract space. Following the EVM, the only other relevant platform that could be found was Neo with the NeoVM as smart contract engine.

Programming languages. Currently platforms mostly need to be developed in languages specifically tailored to them (Solidity, Vyper, e.t.c.), but the trend in the future will be to switch to general purpose virtual machines (e.g. the stack-based Wasm), which support a variety of general purpose programming languages (C, C++, Rust, .NET languages, Java, Ruby, Go, e.t.c.).

CHAPTER 3

Ethereum

This chapter dives into details of the smart contract platform Ethereum. Aspects that are important in later chapters, like the structure of messages and the serialization format RLP, are depicted in greater detail. Areas less relevant for this thesis, as the interaction between nodes or the JSON-RPC API, have been omitted. For those parts we redirect the reader to the comprehensive Ethereum Wiki[84], that deals with every aspect in detail.

The backbone of this platform is the *Ethereum Virtual Machine* (or *EVM* in short), which is a decentralized, turing-complete smart contract virtual machine that executes the bytecode of contracts using an international network of public nodes. The bytecode is generated by a compiler (e.g. *solc*) from human-readable program code written in a variety of supported programming languages, like the commonly used *Solidity*, and the now deprecated lower level language *Serpent*, or its successor *Yvper*. While thy syntax of *Solidity* is influenced by *JavaScript*, the syntax of *Vyper* looks similar to *Python*.

Any valid transaction in the network gets published to all nodes in the network, and the code of each transaction is executed by all nodes to either mine a new block, and therefore generate a new "world state", or by validating nodes to verify the new blocks. Therefore Ethereum is often described as a "world computer"[67].

We provide real-world examples for the structures shown here, that are implemented using Py-EVM (which gets introduced in detail in chapter 5). The source code for the examples can be found in Appendix B.

3.1 Definitions

EIP The Ethereum standard, including core protocol specifications, client APIs, and contract standards, is defined by the *Ethereum Improvement Proposals*. They are pub-

licly available in Ethereum's EIPs repository[1].

Keccak-256 Hashing function used in Ethereum; similar to SHA3-256, only with different padding.

externally owned accounts Ethereum accounts, which are controlled by users private keys

contract accounts Ethereum accounts, which are controlled by contract code

state The state consists of accounts with a 20-byte address.

3.2 Forks

Ethereum is still actively developed. This make changes to the protocol necessary from time to time, which can be implemented via *Soft Forks* or *Hard Forks*. *Soft Forks* are backwards compatible to the previous protocol, and don't necessarily require updates of the nodes because blocks on the forked chain still follow the consensus rules of the old chain as well as new ones. *Hard Forks* are backwards-incompatible and therefore require all network participants to upgrade to a new version of the software.

When talking about the execution of smart contracts, it is therefore necessary to mention under which set of rules (under which forks) this execution happens. Clients that want to verify the whole mainnet of Ethereum need to implement all forks, as they get activated at specific blocks in the chain.

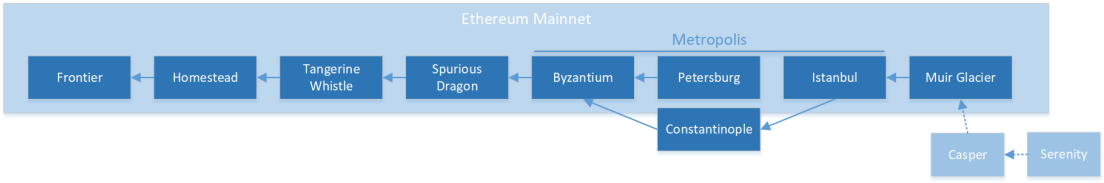

Figure 3.1: Hierarchy of Ethereum Forks, as implemented by the Py-EVM client (light blue shows still unimplemented, future forks)

Figure 3.1 shows the currently active forks of the mainnet, as well as two proposed future upgrades[2].

Frontier was the first version of the Ethereum protocol (released after the *Olympic testnet*), starting at block 1.

[1] https://github.com/ethereum/EIPs

[2] This graphic had to be updated multiple times during the progress of this thesis, as development of Ethereum is progressing fast. As after the publication it probably has changed again, we recommend checking the sources for more up-to-date information.

Homestead, the first major stable upgrade to *Frontier*, triggered at block number 1,150,000 with EIP-2[3], included updates to the transaction processing, gas pricing, security, etc. This fork also contains the logic for the infamous rollback of the blockchain, that was necessary after the hack of the Distributed Autonomous Organization (DAO) in July 2016 and resulted in a theft of 3.6 million Ether.

Tangerine Whistle implemented EIP-150[4], which changed the gas costs for certain operations as a result of ongoing denial-of-service attacks.

Spurious Dragon was another security update, implementing EIP-161[5], that again changed the gas cost to combat DOS attacks, enabled state clearing to clean up empty accounts, and included a security feature that prevented the replay of transactions from the *Ethereum Classic* blockchain on the mainnet of Ethereum.

Byzantium[6] was the first step of next next major platform upgrade *(Metropolis)*. It contains several updates, including a reduction of the block reward[7], the *REVERT* instruction, which allows error handling without consuming all gas[8], zk-SNARKS and other cryptographic primitives[9], as well as several new opcodes. It also delayed the "Ice Age difficulty bomb", which was meant to encourage a fast upgrade to the proof-of-stake-based Ethereum 2.0, as no such solution was expected to be available short-term.

Constantinople[10], the hard fork to activate the second part of *(Metropolis)*, included changes to fee structure, as well as highly anticipated improvements to the scaling of the platform. With this upgrade, also the difficulty bomb got defused again by postponing it.

St. Petersburg activated on the same block as *Constantinople* (on the mainnet) and was necessary to reverse a change of the *Byzantium* upgrade, as it contained a vulnerability which could have been used for reentrancy attacks[11].

Istanbul[11] was the final part of the three-part *Metropolis* upgrade that got released in December 2019. As most other upgrades, it contains changes for the gas metering (e.g. multiple calls to the same *SSTORE* region become cheaper), and added an opcode to get the current chain ID (for more performant layer 2 solutions)

Muir Glacier[12] was an emergency fork to further delay the "Ice Age difficulty bomb" for a third time.

[3] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.md
[4] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md
[5] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-161.md
[6] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-609.md
[7] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-100.md
[8] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-658.md
[9] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-196.md, https://github.com/ethereum/EIPs/blob/master/EIPS/eip-197.md
[10] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1013.md
[11] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1679.md
[12] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2387.md

In a more distant future will be the *Casper* and *Serenity* upgrades, which should completely revamp Ethereum, and include a new virtual machine using web assembly (eWASM), sharding chains, and a switch from Proof-of-Work consensus to Proof-of-Stake.

3.3 Addresses

Addresses in Ethereum are derived from ECDSA private keys. They use the same elliptic curve for the public-key cryptography as Bitcoin, namely the standardized *secp256k1* curve. This elliptic curve is defined by the equation $y^2 = x^3 + ax + b$. In the case of secp256k1 the constant a is 0, and b is 7.

To begin with the address derivation algorithm, we need to (1) create a valid private key. In Ethereum this can be any 32 byte long value, although in reality one should also take sufficient randomness into account. For exemplary purposes, our private key is: *0xc50de8a23afb2dd1fd05b5a79cf19d67c4949bc7857e965c2b9ef32794e14388*.

From this private key we can (2) derive the 64-byte long public key. This derivation is done by applying the ECDSA to our private key. The results of this operation are the two 32-byte long integer coordinates X and Y of a point on the elliptic curve. Concatenated together, they result in the 64-byte long private key. In our example this would be *0x142569909dc3e8d74655e629213cf4010b7a5ed2b20aa723c7671f3e5a9fac793e15bbc7e0da8702d493ddfc9490f3f6379c2a5038ce8ea05bae69485a5dae7b*.

Finally we only need to (3) derive the address from the public key via the Keccak-256 hash function and drop the first 12 bytes of the output. The address is usually represented via 40 hexadecimal characters, starting with '0x', in our example it is *0xbecca0dad3f3a8095e450d1de9cdd7f18cf077af*.

EIP 55 defines a method to include a checksum with the address. This is done in a rather clever way, based on the fact that the address representation is just made up of bytes, and therefore their casing does not matter. So by encoding the address in such a way that if the ith digit is a letter (ie. it is one of abcdef) it is printed in uppercase, if the $4 * i$th bit of the hash of the lowercase hexadecimal address is 1 otherwise it is printed in lowercase. This method keeps the address length at 40 characters and is backwards-compatible to non-checksummed addresses. For our address, this results in *0xBecca0DaD3F3a8095e450d1DE9cDD7f18cF077Af*.

Figure 3.2 shows an example of the address derivation process. It assumes that a person, called Becca, starts with a random private key, and ends up with the vanity address[13] *0xBecca0DaD3F3a8095e450d1DE9cDD7f18cF077Af*. This address, shortened to 0xBecca, is used for exemplary purposes throughout this thesis.

[13] a vanity address is an address that starts with certain letters that spell out names or brands

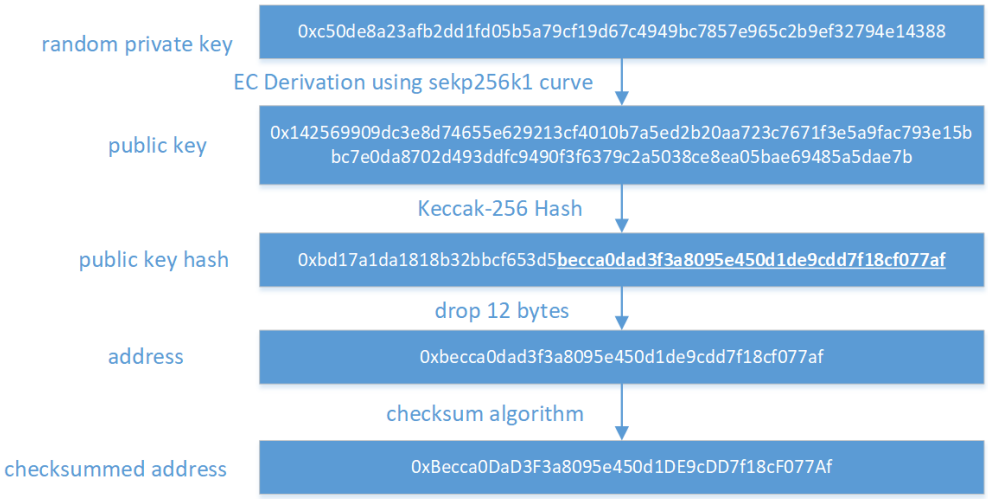

Figure 3.2: Address derivation

3.4 Accounts

At each address resides an Ethereum account. This data structure is made up of the following fields:

- a **nonce**, which is a counter that is increased for every transaction that is sent from this account,
- a **balance**, the amount of ether that resides at this address,
- a **storage root**, which is the merkle hash of the accounts storage, by default the blank root hash and
- a **code hash**, for the hash of the code on contract accounts, or the empty hash on externally owned accounts

3.5 Contracts

Contracts are accounts, which have a non-empty contract code stored. They are therefore not controlled by users, but rather by code.

An important distinction to make here is between transactions and messages. *Transactions* are signed data packages, which are explicitly recorded in the blockchain. Internally in the EVM execution environment those transactions generate *Messages*, which represent the byte data and Ether value that gets passed on between two accounts. As those messages only exist internally in the EVM, they are not recorded on the blockchain.

0x	0	1	2	3	4	5	6	7
	8	9	a	b	c	d	e	f
0	STOP	ADD	MUL	SUB	DIV	SDIV	MOD	SMOD
	ADDMOD	MULMOD	EXP	SIGNEXTEND				
1	LT	GT	SLT	SGT	EQ	ISZERO	AND	OR
	XOR	NOT	BYTE	SHL	SHR	SAR		
2	SHA3							
3	ADDRESS	BALANCE	ORIGIN	CALLER	CALLVALUE	CALLDATALOAD	CALLDATASIZE	CALLDATACOPY
	CODESIZE	CODECOPY	GASPRICE	EXTCODESIZE	EXTCODECOPY	RETURNDATASIZE	RETURNDATACOPY	EXTCODEHASH
4	BLOCKHASH	COINBASE	TIMESTAMP	NUMBER	DIFFICULTY	GASLIMIT		
5	POP	MLOAD	MSTORE	MSTORE8	SLOAD	SSTORE	JUMP	JUMPI
	PC	MSIZE	GAS	JUMPDEST				
6	PUSH1	PUSH2	PUSH3	PUSH4	PUSH5	PUSH6	PUSH7	PUSH8
	PUSH9	PUSH10	PUSH11	PUSH12	PUSH13	PUSH14	PUSH15	PUSH16
7	PUSH17	PUSH18	PUSH19	PUSH20	PUSH21	PUSH22	PUSH23	PUSH24
	PUSH25	PUSH26	PUSH27	PUSH28	PUSH29	PUSH30	PUSH31	PUSH32
8	DUP1	DUP2	DUP3	DUP4	DUP5	DUP6	DUP7	DUP8
	DUP9	DUP10	DUP11	DUP12	DUP13	DUP14	DUP15	DUP16
9	SWAP1	SWAP2	SWAP3	SWAP4	SWAP5	SWAP6	SWAP7	SWAP8
	SWAP9	SWAP10	SWAP11	SWAP12	SWAP13	SWAP14	SWAP15	SWAP16
a	LOG0	LOG1	LOG2	LOG3	LOG4			
b								
c								
d								
e								
f	CREATE	CALL	CALLCODE	RETURN	DELEGATECALL	CREATE2		
			STATICCALL			REVERT	INVALID	SELFDESTRUCT

Table 3.1: Complete list of opcodes at the time of *Constantinople* hard fork, data from [20]

The EVM is a stack-based machine, which means that the operands of the instructions are held in a virtual stack, instead of registers. As they are in a known location on the top of the stack, the instructions require no memory addresses or register numbers for their operands.

Besides the stack memory, the EVM also has the ability to load data from and store data in volatile memory. The account storage is the persistent memory in a key-value store.[81]

The bytecode of contracts is executed by the EVM. It supports over 100 instructions (opcodes), including the typical arithmetical and logical operators nearly every VM provides (ADD, MUL, AND, OR, EQ, etc.), memory and storage manipulation (MSTORE, SSTORE, MLOAD, JUMP, etc.), but also natively supports cryptographical primitives (SHA3) and Ethereum-specific instructions (BALANCE, GASPRICE, BLOCKHASH, COINBASE, etc.). Because of the stack-based nature of the EVM, stack-manipulation opcodes (POP, PUSH, DUP, SWAP) are available as well. The opcode SELFDESTRUCT allows a contract to self-destruct, and send the remaining ether value to another account.

Table 3.1 shows the complete list of currently implemented and still undefined opcodes.

Table 3.2 describes several selected opcodes and their semantics in detail. We can see

Opcode	Mnemonic	Stack Input	Stack Output	Expression	Notes
01	ADD	a | b |	a+b |	$a + b$	(u)int256 addition modulo 2**256
80	DUP1	value |	value | value |	$PUSH(value)$	clones the last value on the stack
20	SHA3	offset | length |	hash |	$hash = keccak256($ $memory[offset : offset + length])$	keccak256

Table 3.2: Description of selected opcodes from ethervm.io[20]

that, for example, the opcode for *ADD* takes the two values on top of the stack, adds them, and then pushes the result back to the top of the stack.

The opcode for *DUP* clones the last value on the stack. To save some space, several different variants (*DUP2*, *DUP3*, ...) of this opcode exist, they only diverge in the value that gets cloned (2nd last, 3rd last,..) and is pushed to the stack.

Finally there is a more complex instruction with the mnemonic *SHA3*, which takes an offset and a length parameter from the stack, and calculates the Keccak-256 hash value of that memory region.

Vaibhav[81] provides a detailed overview on how the EVM works in the background, the machine space it provides, and how the execution model works.

As EVM smart contracts are turing complete, there are no limitations to the possible usages of such contract code. But a recent study has confirmed that the main areas of applications are still tokens (and their ICOs), games and gambling, exchanges and wallets - at least of those 0.2% of contracts, that have ever been called.[19]

3.5.1 Gas

In contrast to IOTA, users of Ethereum need to pay fees for their transactions. This fee is measured in *Gas*, and every operation requires a certain amount of gas. The intent of this fee system is to limit denial of service attacks on the network, as attackers would have to pay proportionately for the computation, bandwidth and storage they consume. This fee is then given to the miner of the block, whose block includes this transaction.

The price per computational step is not static, instead a value for the *gasprice* has to be specified by the sender, that represents the fee the sender pays per computational step. This allows for complex gas economics, where miners can decide the minimum *gasprice* they are willing to accept. It means that the gas price is subject to supply and demand - in periods where there is much network activity and therefore a large number of transactions competing to be included in the blockchain, the gas price will therefore be higher than in periods with low network activity.

Users that execute a contract send the maximal amount of gas they are willing to pay alongside the transaction, any unused gas will be refunded. If the supplied amount of gas was too low, the transaction will be reverted.

3.5.2 Tokens

Because tokens are so heavily used on Ethereum, our example code follows that design pattern as well. Tokens can be seen as digital representations of assets (like vouchers, IOUs, or real-world, tangible objects). In Ethereum, they are smart contracts, that keep track of the balances of accounts in the contracts storage.

At present, more than 200.000 contracts[14] are already deployed, following the most commonly used ERC-20[15] technical standard for tokens. This specification defines the methods *name*, *symbol*, and *decimals*, which return constants describing the token, *balanceOf* to get the balance of a specified address, *transfer*, *transferFrom*, and *approve* to allow the transferring of tokens between people or contracts, and *allowance* and *totalSupply* as view-functions. Because most of the tokens follow this or a compatible standard, wallets and exchanges only need to implement this interface to be able to support the majority of tokens.

The full interface specification ERC-20-compatible smart contracts need to implement can be found in listing 8.

3.5.3 Transaction

Transactions in Ethereum contain the following fields:

- **nonce**: the value of the increasing transaction counter
- **gas price**: representing the fee the sender pays per computational step
- **gas**: the maximal fee the sender is willing to pay
- **to** address: the recipient of the message
- **value**: the amount of ether to transfer from the sender to the recipient (which can be zero)
- binary **data**: the data that is sent with the transaction (which can be empty)

Signed transactions also contain a cryptographic signature (in the fields v, r and s) identifying the sender.

3.5.4 Contract creation

New contracts are written to the blockchain via deployment code. The creation happens via a transaction to the *create contract address*, which is the address that contains only zeros (0x0).

[14] https://etherscan.io/tokens
[15] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

The data of this transaction is the deployment code. This contains the opcode *CREATE* (0xF0) - or since the Constantinople Upgrade, the opcode *CREATE2* (0xF5). These methods basically execute the constructor of the to be deployed contract, and then return the rest of the bytecode without the constructor, which then is written to the blockchain.

The only change between the *CREATE* and *CREATE2*[9] opcodes is the address, on which the contracts gets deployed.

For the standard *CREATE* call, the address is simply a hash of

- the sending account address, and
- the sending account's current nonce

This ensures that different accounts will generate different contract addresses, and different transactions of the same account also generate different addresses, even if the contract bytecode is the same.

The parameters, that the *CREATE2* call uses, are

- the sending account address, and
- the contract's initcode (which gets executed to generate the runtime bytecode that will be placed into the state), and
- a salt, chosen by the developer

The motivation for adding this opcode was to be able to deterministically know a contracts address, even before the contract code is deployed. The code, that a particular address will eventually contain, can only come from a particular initcode.

As the deployment code can contain arbitrary instructions, its behavior can be completely different than what is expected from a "plain contract creation". It is even possible that this deployment code creates other contracts, or even already self-destructs in the deployment. This strange behavior is actually very common in the mainnet of Ethereum.[19]

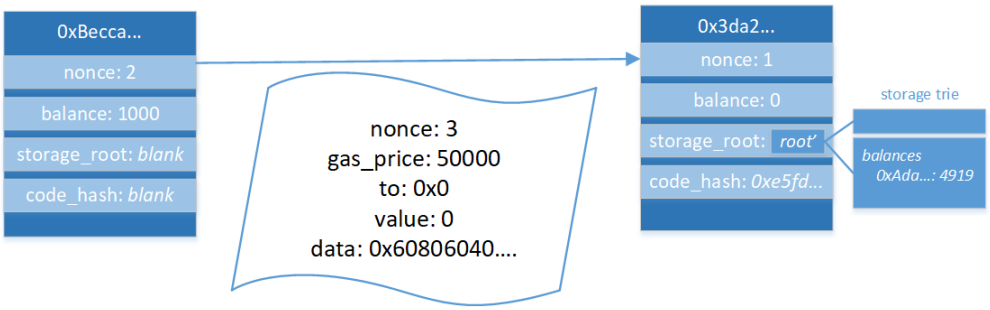

Figure 3.3: Create Contract Transaction

Figure 3.3 shows an example for a contract creation. Becca deployes a new token contract (let's call the token *SuperToken*) on the blockchain - its source code can be found in listing 7. This is done by sending the deployment code for the contract to address 0x0. Based on the Becca's address and the nonce of the transaction, the contract's code gets stored on a new account (0x5643f...).

Figure 3.4: Contracts

3.5.5 Contract interaction

Users interact with contracts using their Application Binary Interface (ABI). Method calls to contracts also happen via EVM transaction, but now the recipient is directly the contracts address. Usually the interface functions are static and known at compile time, therefore the data of the transaction only needs to contain the function selector and the parameters.

Getting back to our example, let's suppose Becca wants to send all of the 4919 tokens, which were initially minted to her address, to her friend Ada (see figure 3.4 for a high-level overview of that transaction).

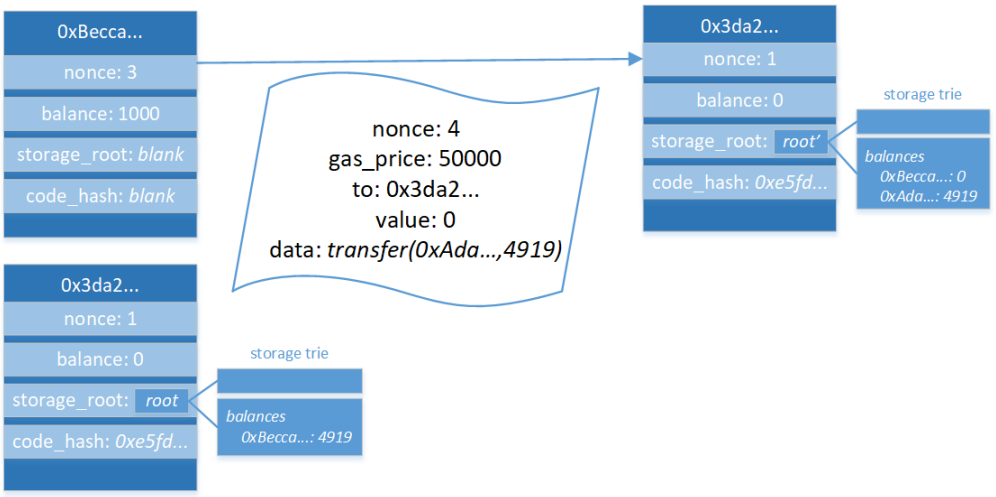

Figure 3.5: Contracts

We already know that the token balances are not properties of the Ethereum account itself, but rather something the deployed smart contract keeps track of on its own in its storage. Therefore, when Becca wants to send tokens to Ada, she doesn't have to interact with Ada directly, but only with the *SuperToken* smart contract (see figure 3.5). She has to call the *transfer* method with the desired amount of tokens she wants to send to Ada (when sending tokens to another smart contract, the interaction would be slightly different), the contract will then update its register in the storage to deduct the tokens from Beccas address, and credit them to Ada in an atomic operation.

Function selectors

Unlike to static programming languages like C, where the symbol table in a shared library contains the name and types, calls in Solidity are made via a function selector. Each method in Solidity is a four byte long jump destination, which is generated from the first four bytes of the Keccak-256 hash of the signature of the function. The signature is defined as the canonical expression of the basic prototype without data location specifiers, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma - no spaces are used.[14]

As an example we take a look at the *transfer* method of token contracts. The signature of this function is $transfer(addressto, uint256amount)$, and therefore the function selector is $a9059cbb$.

$$keccak256(\texttt{transfer(address,uint256)}) = \underbrace{\texttt{a9059cbb}}_{\text{first 8 chars}}\texttt{2ab09eb219583f4a59a5d0623ade346d962bcd4e46b11da047c9049b}$$

Because the function selector is merely a hash, it is not possible to reconstruct the original names of methods from the compiled contracts on the chain. Therefore the compiler not only generates the binary code of the contract, but also a JSON-encoded ABI file, which contains metadata (including the name and parameters of methods and events). This file is then given to the users of the DApp (either directly to load the ABI and the contract address into a local *geth* client or a web wallet; or by providing a web interface for their DApp that makes use of the ABI), to allow them to conveniently interact with the app[16].

There are also repositories available, that contain the function selectors of well-known functions[17], where e.g. the *transfer* function can be found. It is also possible that different methods result in the same selector[18].

Function arguments

After the function selector, the encoded arguments follow. For details about the encoding, we redirect the reader to the Contract ABI Specification[14] - for now, it is only important to know that certain elementary types exist (like *uint256*, a 256 bit long unsigned integer; or address - which is equivalent to *uint160*). Parameters are padded in such a way, that their length is a multiple of 32 bytes (for *uint* zero-bytes are added on the higher-order (left) side).

For our token transfer example, the first parameter is the *to* address. As second parameter follows the amount of tokens in hex, which is 4919_{10} = `0x1337`.

Combining the function selector with the parameters results in the following transaction data (the line breaks are just for ease of reading):

```
a9059cbb                                                                  } transfer
0000000000000000000000ada5547578a08c7d991811c83616b7f3ed33b795            } to
0000000000000000000000000000000000000000000000000000000000001337         } amount
```

3.6 Recursive Length Prefix

It is intuitive to say that Ethereum stores transactions in a blockchain, but this only serves as a high level description. When looking into how transactions are actually transmitted or stored, a deeper understanding of the used data serialization is necessary.

In Ethereum the Recursive Length Prefix (RLP) encoding is used throughout the project. For example, when a client wants to make a transaction, it encodes the transaction

[16] users still need a verified source code of the contract, so they can independently verify that the implemented functions really match the specification

[17] e.g. https://github.com/ethereum-lists/4bytes

[18] in the CTF challenge *Acoraida Monica* of the Real World CTF 2018 a contract was deliberately crafted to exploit this behavior[51]

object as RLP byte array, then signs this data, and submits the RLP-encoded signed transaction to its peers. But it is not only used in the network layer, all common Ethereum implementation also use it for serializing entries in the database (more of this use case will be shown in section 6.5.4).

While serialization formats like JSON or XML could have been used as well, the Ethereum developers designed a format that is highly minimalistic and space-efficient, and which can be deserialized again quickly (unlike JSON or XML, which tend to be on the verbose side).

RLP natively supports arbitrarily nested arrays of binary data. The representation of data in more useful data types (like strings) has to be handled by higher-order protocols. It only encodes the structure of the data, but doesn't know anything about the kind of object it was before. While this reduces the overall size of the encoding, it requires the decoder to know what kind of object it is looking, to be able to make any sense out of it.

The RLP encoding is actually quite simple. It can only encode items, and differentiates between two possible data inputs.

An item is defined as

- a string, or
- a list of items

All other data types can be converted to strings (or more precisely arbitrary byte arrays).

The encoding algorithm outputs the length of the string or list (but in a clever space-efficient way, which favors short strings) first, followed by the data itself.

The rules for the encoding algorithm are defined as:

- `[0x00, 0x7f]` for a single byte in the `[0x00, 0x7f]` range
- `[0x80, 0xb7]` for strings with a length between 0 and 55 bytes. The first byte is the length of the string, added to $0x80$. Then the string itself follows.
  - `[0x80]` is the zero-length string, identical to non-values like $uint(0)$, $string("")$ and empty pointers
- `[0xb8, 0xbf]` for strings with more than 55 bytes. The first byte is $0xb7$, added to the length in bytes of the length of the string in binary form, then the length of the string follows, and then the string itself.
- `[0xc0, 0xf7]` for lists with a complete length (the combined length of all its items being RLP encoded) between 0 and 55 bytes. The first byte is the complete length, added to $0xc0$. Then the RLP-encoded items are concatenated.

  - `[0xc0]` is the empty array `[]`

- `[0xf8, 0xff]` for lists with a complete length of more than 55 bytes. The first byte is $0xf7$, added to the length in bytes of the length of the payload in binary form, then the length of the payload follows, and then the concatenation of the RLP-encodings of the items.

As an example, let us try to encode the token transfer transaction:

- **nonce**: 4
- **gas price**: 13
- **gas**: 5000000 wei
- **to**: 0x5643f85c81eEcECd195D4cC29C9b9877337A1550
- **value**: 0 ETH
- binary **data**: a9059cbb000000000000000000000000ada5547578a08c7d991811c83616b7f3ed33b795000000000000000000000000000000000000000000000000000000000000001337

The **nonce** is just a *uint* between `[0x00, 0x7f]` therefore the encoding is simply the value:

`04` } uint(4)

The **gas price** is also a single *uint*, $13_{10} =$ `0x0d`.

`0d` } uint(13)

The **gas** is the *uint* $5000000_{10} =$ `0x4c4b40`, which is a long string as it has 3 bytes.

`83` } length 3 -> 0x80+0x03
`4c4b40` } uint(5000000)

The **to** address is an *address*, which is equivalent to a string with 32 bytes.

`94` } length 32 -> 0x80+0x20
`5643f85c81eececd195d4cc29c9b9877337a1550` } to address

The **value** is *0*

```
80   } uint(0)
```

The binary **data** is a string of the length 68 (that is more than 55 bytes), and therefore first needs to encode the length of the length of the string (1 byte), then the length of the string (68 bytes), and finally the string itself.

```
b8                                                                 } length of length 1 -> 0xb7+0x01
44                                                                 } length 68 -> 0x44
a9059cbb000000000000000000000000ada5547578a08c                     }
7d991811c83616b7f3ed33b795000000000000000000000                    } data
0000000000000000000000000000000000000000001337                     }
```

To complete the RLP encoding for the transaction, we need to add the total length of the list to the beginning, which is 98 bytes. As this is more than 55 bytes, the first position needs to be the length of the length (1 byte), then the total length of the payload (98 bytes). Afterwards follow the concatenated RLP-encoded items.

```
f8                                                                 } length of length (1) -> 0xf7+0x01
62                                                                 } length of payload (98 bytes)
04                                                                 } nonce
0d                                                                 } gas price
83                                                                 }
4c4b40                                                             } gas
94                                                                 }
5643f85c81eececd195d4cc29c9b9877337a1550                           } to
80                                                                 } value
b8                                                                 }
44                                                                 }
a9059cbb000000000000000000000000ada5547578a08c                     }
7d991811c83616b7f3ed33b795000000000000000000000                    } data
0000000000000000000000000000000000000000001337                     }
```

For a signed transaction the signature with the parameters v, r and s would follow after the *data* parameter.

CHAPTER 4

IOTA

While IOTA is not an abbreviation for the Internet of Things (IoT), but instead comes from the 9th letter of greek alphabet - *iota* - and is a play on the meaning of *small quantity*, the IoT is certainly one of the strongest use cases for this cryptocurrency. Accordingly, feeless microtransactions (e.g. in a machine-to-machine ecosystem, where a car can automatically and seamlessly pay a smart parking meter for every second it uses the parking space[62]) are a core feature.

As already seen in figure 1.6 contrary to Blockchain-based approaches, IOTA does not group multiple transactions into blocks. Every vertex in this graph represents just a single transaction. The features of this cryptocurrency are its scalability (there is no inherent limit of the transaction rate in the network in the protocol[1], leading to claims of an infinite scalability), its decentralization (everyone who publishes a transaction is also validating the preceding transactions, and therefore participates in the consensus), the lack of transaction fees (making it usable for micropayments), and its resistency to being cracked by quantum computing.[65]

At present the native support for smart contracts via the *Qubic* project (see section 2.7) is still under development and no working prototype is yet released, but the possibility for zero-value transfers (containing only data, and no monetary value) alone enables several applications in the IoT space (e.g. a small application we developed with IOTA[34]).

As previously done with Ethereum, the examples in this chapter are also as close to typical real-world scenarios as possible.

4.1 Definitions

Curl Hashing function used in IOTA; uses purely trinary logic; was replaced for most usages by *Kerl* due to found vulnerabilities[33][75]

[1] https://coordicide.iota.org/post-coordinator

Kerl Hashing function used in IOTA; wrapper that converts between trinary and binary representations and utilizes the well-known Keccak-384 hash function[35]

4.2 Trits and Trytes

The first striking derivation of IOTA, when compared to other cryptocurrencies, is the usage of a different numeral system. Instead of using the binary numeral system like in almost all modern computers, IOTA's developers decided to use balanced ternary instead, a numeral system with 3 digits, -1, 0 and 1 (which are often abbreviated as -, 0 and +). Such a digit is called a *trit*. The analogy to a *byte* in this system is a *tryte*, but there is no general consensus of how many trits make up a tryte. One commonly used interpretation is the *tryte3*, which consists of three trits and therefore 27 possible states. Those numbers are then represented with the 26 letters of the English alphabet A-Z, and one additional character (9). For the remaining document this will be the default type for a tryte, when no size is specified.

Another commonly used type is the *tryte6*, which can hold six trits (converted this would be 9.5 bits, and therefore more than a byte).

The reason the IOTA developers went with this system is, that ternary processors are theoretically more efficient than binary processors; and certain mathematical constructs are more cleanly represented in balanced ternary. While heavily criticized for it (e.g. in [56] or [41]), because all computer hardware today uses binary, therefore requiring the conversion to be done in software which is inefficient and complex, the developers still stand by this decision.

4.3 Addresses

Addresses in IOTA have a length of 81 trytes, and can optionally include a checksum of 9 trytes.

The address derivation scheme is shown in figure 4.1. As usual for cryptocurrencies, the private keys are derived from a secret seed. This seed is a 81 trytes long random string. To generate different key pairs, an index number that is incremented for each new key is appended to the seed. For the key generation, alongside the seed and the index, a security level is also needed, which specifies the key length and therefore the security of signatures. Levels 1 to 3 are possible, and Level 2 (with a signature length of 4,374 trytes) is the default. The Kerl-based hash function now takes this seed, index number and security level, and generates the 81-tryte subseed.

From this subseed the private key is derived by absorbing and squeezing it into a sponge function 27 times per security level. The sponge function is a cryptography primitive used in the Keccak (SHA-3) family of hashing functions. The sponge construction is an iterated construction for building a function with variable-length input and arbitrary output length based on a fixed-length transformation or permutation f, operating on a

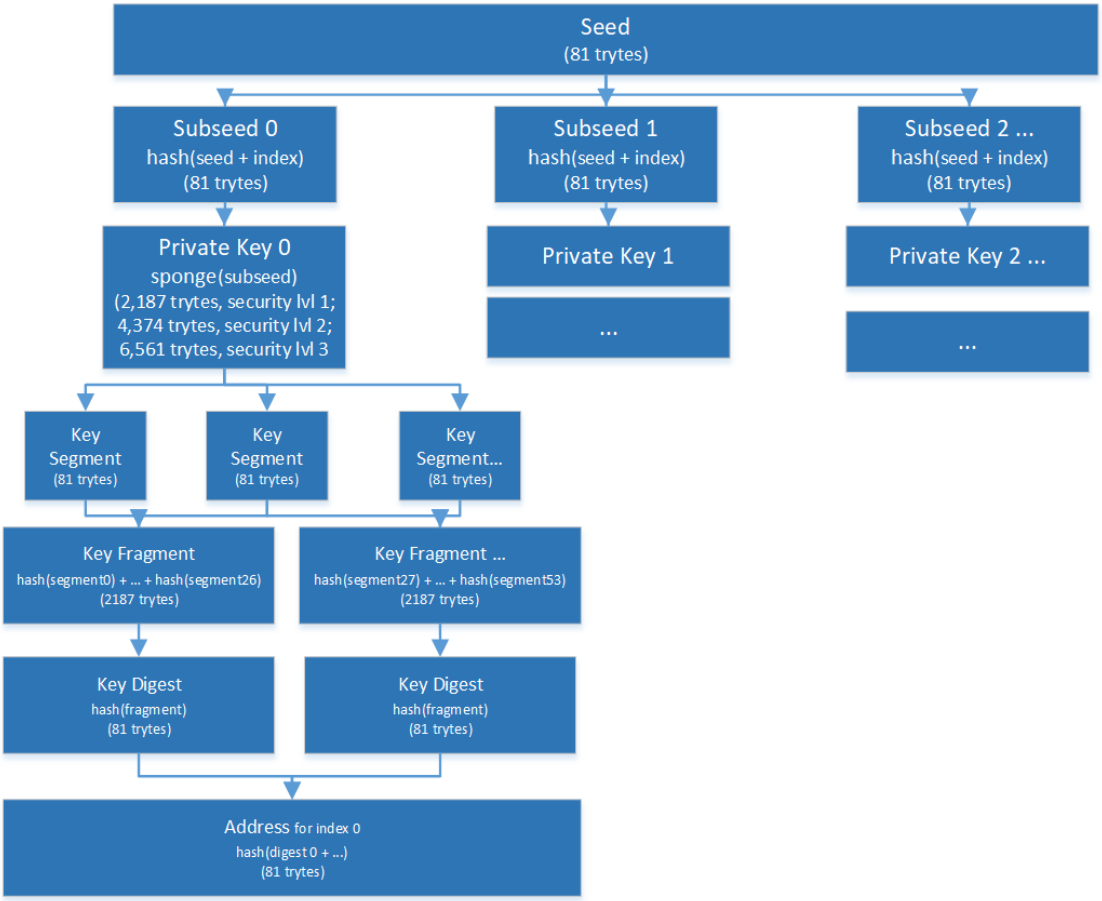

Figure 4.1: Address generation in IOTA

fixed number of bits. During the absorbing phase any amount of data can be inputted, the blocks are XORed into the state, interleaved with applications of the function f. During squeezing, bits of the state are returned as output blocks, again interleaved by f. The number of output blocks can be chosen by the user.[29]

This private key is used to derive the 81-tryte address, by splitting it into 81-tryte segments, hashing each segment 26 times, combing 27 segments again to a key fragment, hashing the fragment once to get a key digest, and finally combining the digests and hashing it one last time.

When, for example, this process is started with the random, but no longer secret and safe, seed `VGTRQMLEREAWMOWVFZEODTZ9VFFLOYLBZCHBGVDPZNVUIWRRNHQPNNQTPZOZSGBU BUCVLUBHUQCHUIIWI`, the address generated for the first index 0 is `BMGGDBRHRNTPRGNTJ VWXESRBUPF9SEHUPYF9HFPHDNSDK9KGLMPAIWHKKOMOINTJHFORULJOUAVPJBHUD`, followed by its checksum `WEETMEWOW`.

4.4 Signatures

IOTA uses the quantum computing-resistent W-OTS. The main reason why this scheme was chosen is its resilience against attacks by quantum computers, which the developers see as a real threat.

But this feature brings two main drawbacks with it: First, any signature generated via this method reveals parts of the private key, that was used to make this signature. If the same key has already been used twice, large parts of it have been leaked, and attackers can possibly already guess (brute-force) the whole private key, and therefore correctly sign a third message. Second, the signatures are quite long (at a minimum 2187 trytes, which is far longer than the 65 bytes with the Elliptic Curve Digital Signature Algorithm in Ethereum).

There was also a weakness found that did not require a key reuse, which has been mitigated by IOTA.[46]

Technical details how the signature algorithm is implemented can be found in a blog post by Wolfgang Welz.[83]

4.5 Transactions and Bundles

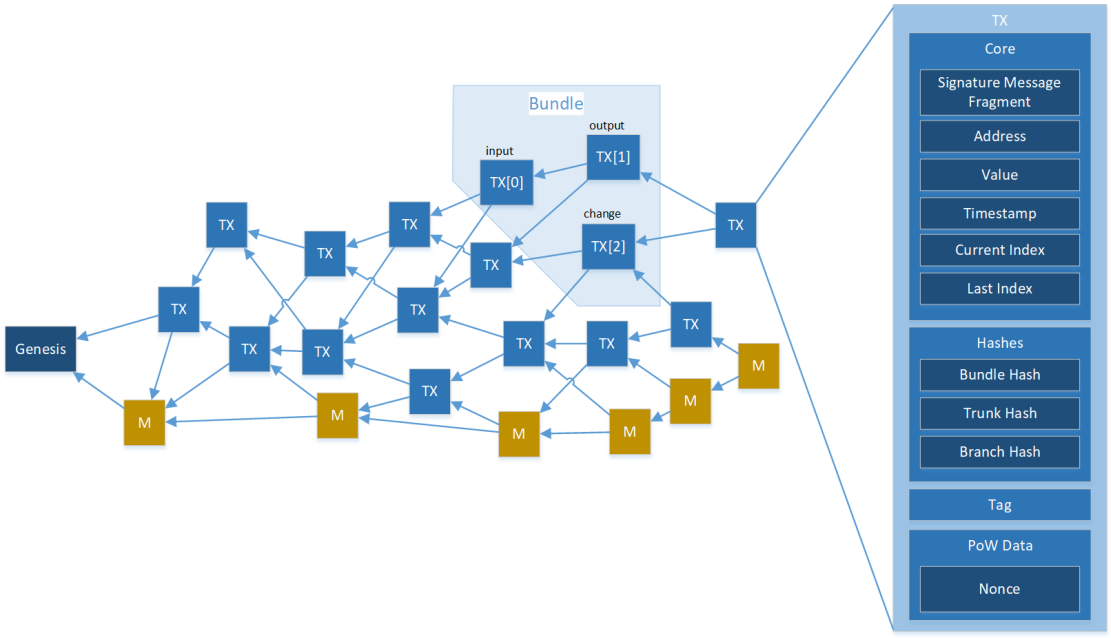

Figure 4.2: Transactions and Bundles

Because IOTA does not have blocks, single transactions are sent to nodes of the network. These transactions are then directly attached to the DAG. IOTA transactions are exactly

2673 trytes long, 2187 trytes of that can be used for either the signature of the transaction or a message. If the size of the signature or message is bigger than this maximum amount for a single transaction, it is fragmented over multiple transactions.

The grouping of those contiguous transactions is called a *bundle*. Bundles are mainly just a logical concept, all transactions of a bundle are still distinct transactions in the ledger. The only physical manifestation of a bundle is in the *bundleHash* field, which uniquely identifies the bundle. Transactions in the bundle are also atomic, either all of them are valid and confirmed, or none of them are.[72]

Figure 4.2 shows three transactions, that are grouped into a bundle. The bundle depicted here is an ordinary transfer of value, consisting of an input transaction withdrawing IOTA tokens from one address. The size of the signature, and following this the number of transactions necessary to fit the signature, in the *signatureMessageFragment* field depends on the security level. With the standard level of 1, the signature can fit into a single transaction, otherwise it needs to be fragmented across further output transactions (where the other fragments are zero-value). The output transaction is sent to the address where the tokens should be withdrawn to. This transaction does not contain a signature (as the sender does not know the private key of the receiver which could make a signature), but could instead optionally contain an arbitrary message in the *signatureMessageFragment* field. And like before, if the message is larger than the maximum field size of 2187 trytes, it can be fragmented over multiple transactions.

Each transaction in IOTA also has an implicit *transaction hash*. This is not an attribute in the transaction object, but simply a *CURLP-81* hash of all trytes of the transaction concatenated.

As addresses can not be reused in IOTA due to the used signature scheme, all tokens need to be withdrawn from the origin address in a transaction. This is similar to how *Unspent transaction outputs* in *Bitcoin* work, but in IOTA this is actually not enforced on a protocol level, but left to the responsibility of clients. They generate another output transaction to a self-controlled address, often called a *change* transaction.

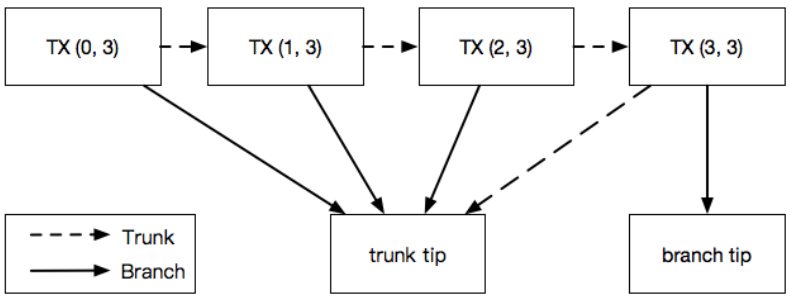

Figure 4.3: Structure of a bundle[76]

Figure 4.3 shows the structure of a bundle. The first transaction in a bundle is called the tail at index 0, the last transaction is called the head, and the remaining ones are the body. A single transaction is head and tail simultaneously.

In IOTA a transaction is linked to two others (the parent transactions), that are verified through this link. One of them is called trunk, the other branch. Starting from the tail of the bundle, all bundle transactions are linked through the trunk hashes, the last (head) transaction links to another transaction in the Tangle. The bundle hash is always a transaction of another bundle, for the tail and body it is the same as the trunk reference from the head, for the head it is a different transaction in the Tangle, selected through the tip selection process.

4.6 Tip selection process and random walks

Every transaction that gets published in the ledger has to approve two previous transactions. The new transaction then adds two edges to those selected transactions. If a transaction has no outgoing edges, and is therefore still not approved or confirmed by any other transaction, the vertex is called a *tip*. Selecting the two tips for approval is done via the *tip selection process* by full nodes.

This process runs the *tip selection algorithm*, which, in its simplest form, puts a walker on the genesis transaction, and walks it towards the tips. As there are no loops in a DAG, it is guaranteed to always end up at a transaction with no outgoing edges, a so-called tip. This process is called *unweighted random walk*. By introducing a *Markov Chain Monte Carlo technique*, that is introducing some bias to avoid older transactions *(lazy tips)* by taking the *cumulative weight* into account, the algorithm gets refined to a *weighted random walk*.[22][23]

4.7 Consensus

This section gives a detailed overview of how consensus is formed in the Tangle. This is of particular interest for this thesis, because later on a similar method will be used to achieve consensus in our prototype for an EVM implementation for the Tangle.

For every cryptocurrency the usage of a suitable consensus method is an important choice, because consensus is what allows the nodes within the network to agree on which transaction is valid within the ledger. In a distributed ledger contradictory statements, like double-spend transactions, or spending from non-exist funds, can exist, and therefore the network as a whole needs to converge to some consistent, contradictory-free state.

An important distinction to make is between the method, which is used currently to achieve consensus in the network, and the plan by the *IOTA Foundation* on how this will be handled in the future.

The currently used method is a fairly simple, although centralized one: A service run by the *IOTA Foundation*, called the *Coordinator* periodically issues a special signed bundle,

called a *milestone*. Per definition, any transaction directly or indirectly referenced by the milestone is immediately confirmed. The coordinator will only reference transaction paths without contradictory statements, otherwise all other nodes in the network would not accept the milestone.

The bundle consists of multiple transactions, which contain the fragmented signature. Due to the increased security level of the signatures, they do not fit into a single transaction. They also work a bit differently to value transaction signatures: A merkle tree of the coordinators public/private keys is generated in advance, and the address of the coordinator is the hash of all leaves of this tree.[1]

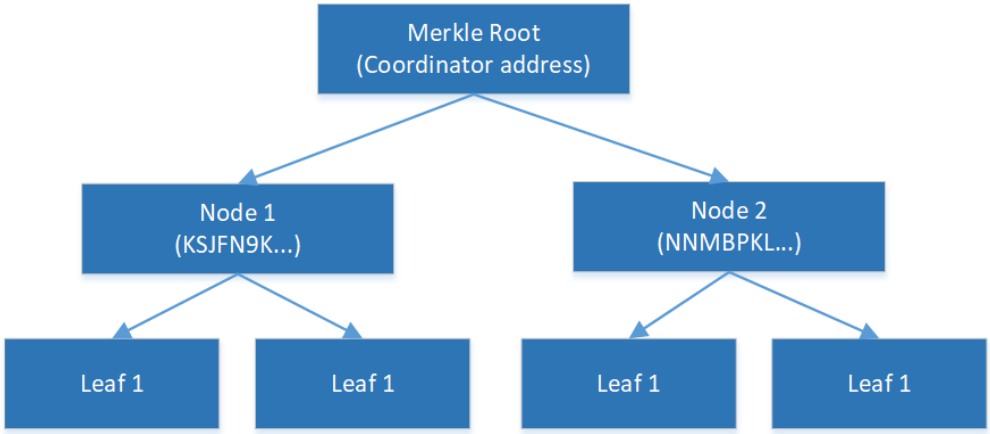

Figure 4.4: Example for a coordinators merkle tree[1]

Figure 4.4 shows an example for such a merkle tree. The repository *iota-community/one-command-tangle*[2] contains an exemplary precomputed tree with a depth of 20, which can issue 1073741824 milestones (at a rate of one milestone per 30 seconds, this lasts over a year).

To verify the milestone, nodes rebuild the merkle tree, and check if the rebuilt merkle root matches the coordinator address. If this is the case, and the milestone otherwise does not contain any conflicting information (like double spends), it becomes a valid milestone, and any transaction directly or indirectly referenced by it is confirmed.

The source code of the active coordinator instance running on IOTAs Tangle is available under an open-source license[3].

While often compared to a similar implementation in Bitcoin, namely *snapshots*, which were used in the early days to hinder attacks, it is often criticized for the control the IOTA Foundation has on the network: it can select which transactions are prioritized,

[2] https://github.com/iota-community/one-command-tangle/tree/master/layers
[3] https://github.com/iotaledger/compass

or completely block certain addresses. It is also a single point of failure, because the network would come to a halt if the coordinator stopped issuing new milestones[4].[24][25]

4.8 IXI modules

As the IOTA protocol will not be able to change in the future, IOTA eXtensible Interface (IXI) modules will enhance the API of IOTA to provide specialized functions. Those are best-practice solutions to solve common problems, like utilizing the optional transaction data field to provide message streams (MAM.ixi; Masked Authenticated Messaging), providing a proxy for other programming languages (Bridge.ixi), or finding confidence intervals when certain transactions were published in the DAG (Timestamping.ixi).

[4] bugs in the IRI caused exactly such a failure already[52]

CHAPTER 5

Smart contracts in a DAG ledger

In this section, we try to combine the advantages of smart contracts and DAG ledgers.

As already stated in chapter 3, smart contracts are contracts written in code, that enforce the negotiation or performance of a contract without third parties. Those are features of many second-generation blockchains, most notably Ethereum, that offer a nearly Turing-complete language in their platform. We also already saw that such platforms often suffer from scalability problems, with bottlenecks on transaction throughput when apps like CryptoKitties gain popularity.

DAG ledgers tackle the causes for the scalability problem by getting rid of the linear block chain structure, and arranging the transactions in a graph.

We already have discussed that smart contracts (usually) need total ordering, and are therefore (generally) incompatible with such data structures.

Nevertheless, we propose a compromise in this chapter.

5.1 Relaxations

We already know, that transactions in a DAG are only a partially ordered set. They lack the Totality axiom from a total ordered set – for any two given transactions A and B, we therefore can not say with absolute certainty, which of them happened first.

But, we can argue that we dont even need a (global) total ordering of all transactions. Lets have a look at the following contract interactions:

We can see that different contracts rarely interact with each other (the same is true for a real network, like the Ethereum Mainnet). So, every DApp can build its own chain – completely independent from one another. We dont need to care whether the $breedWith$ method call of the $CryptoPenguin$ contract happened before – or after – the transfer call in the $SuperToken$ contract.

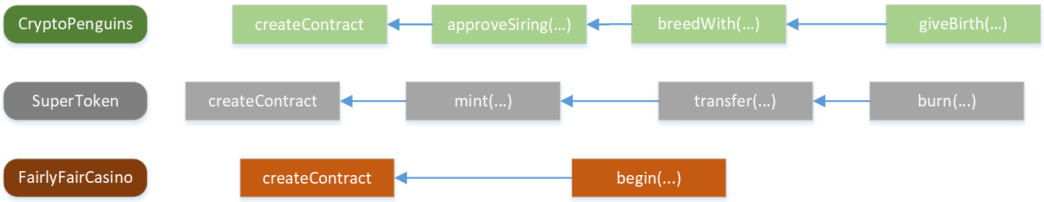

Figure 5.1: Three different contracts on a Blockchain; with their respective method invocation sequence

This idea is not novel – such approaches are called *sidechains*, *bridgechains* (e.g. in *Ark*), or *smart child chains* (e.g. in *Ardor*), as we have already established in chapter 2.9. Our design closely follows the findings of that chapter, where e.g. the article "Solving scalability of Ethereum through Loom Sidechains"[71] describes a similar approach.

Summarized, this allows us to model the invocations of smart contracts in a Directed Acyclic Graph (compare this to the transaction in NANO's block lattice[54] – they look very similar).

5.2 Platforms

In the last section we have discussed how smart contracts could be embedded into a DAG ledger structure. Now we want to see how this could work in practice. For that we need two ingredients: a Smart Contract Virtual Machine, and a DAG platform that we can use to piggyback the smart contract VM.

Following the arguments of section 2.6, we can also rule out the use of Neo, arguably the strongest contender for Ethereum, as our smart contract platform of choice. The NeoVM does not provide any clear advantage over the EVM (we did focus only on the computation engine, as theoretical transaction throughput does not matter when deploying the contracts in a DAG), but also the lack of good documentation is a big detriment.

The research in chapter 2 has shown that the Ethereum VM is the most widely used smart contract platform by far - even many other cryptocurrencies use their VM to implement smart contracts. Since we have chosen this EVM engine for our implementation, all existing smart contracts can be reused with little or even no adaptation at all in our DAG chain.

Our goal is to build a second-layer solution for an already established DAG cryptocurrency. Fortunately the choice here is easy, as only two (popular) platforms exist: IOTA and NANO. While the block lattice structure of NANO would look like the ideal choice, the platform needs to support zero-value data transactions. This rules out NANO, and therefore our data storage is on IOTAs *Tangle*.

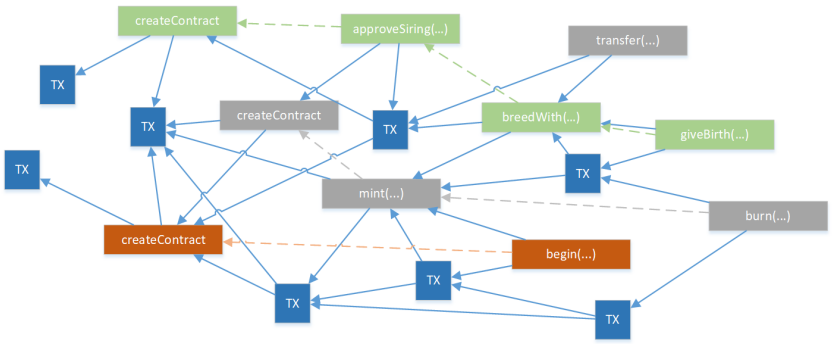

Figure 5.2: Example of contract invocations in the Tangle e.g. there is no direct path between mint() and approveSiring(), so we dont know which TX happened first but we dont need to.

We know that IOTA will implement its own type of smart contract support under the name *Qubic* (as discussed in section 2.7). While this is certainly a welcomed addition, and great for certain tasks, it currently seems likely it will not be as powerful and universally usable as the EVM. It also follows different paradigms, like a functional programming style, which further differentiates it from current smart contract platforms. This obviously does not mean that *Qubic* is a bad idea, just that there is enough space for both solutions to coexist, giving smart contract developers more choices.

A Proof Of Concept shows how EVM transactions can be integrated in the tangle. Users can publish EVM messages (contract creations, or executions) on the Tangle. To ensure some global order of execution, we introduce an EVM coordinator. The job of this service is to determine the execution order of the contracts, and ensure that there is a global consensus about their state. Its implementation is fairly straightforward, as the milestones for this service are basically the same as blocks in Ethereum.

The difference to traditional smart contract blockchain platforms is that due to the graph structure in the ledger, an arbitrary amount of parallel chains can be built by different coordinators. That means, whenever a DApp gets too big and generates a big portion of the traffic of the chain, it can be detached from that chain, and live in its own. This approach is similar to the concept of sidechains, bridgechains[1] or smart child chains[2], and follows in the footsteps of *RSK*, which have all already been briefly discussed in section 2.9.

The advantage of this approach is that different coordinators can follow arbitrary protocols, or even different consensus mechanisms. There could be a chain with EVM smart contracts, right next to a completely independent one containing *NeoVM* bytecode. Out of scope for this thesis is how such chains could interact with one another, but cross-chain interoperability is currently a much researched topic, and various solutions have

[1] as in the *Ark* platform whitepaper, https://ark.io/Whitepaper.pdf
[2] as in the *Ardor* platform whitepaper

been proposed that could be adopted here as well. As a last resort, custom extensions can define new opcodes, that are used for cross-chain message calls.

The only other project we could find, that follows a similar approach is *VITE*. This project builds a complete, independent ledger, with a structure similar to the *block lattice* of *NANO*[3]. We argue that our approach to leverage an already existing and well-tested ledger and build a second-layer technology on top of that is a better solution.

[3]Due to the usage of a *snapshot chain*, a globally unique order of transactions is introduced again. Although this snapshot chain references the other transactions in the DAG, but left by itself it behaves like a traditional blockchain.

CHAPTER 6

tanglEVM - an EVM in the Tangle

We have chosen IOTAs Tangle as ledger to piggyback our smart contract platform for multiple reasons: The first requirement for the ledger is the support of zero-value data messages. While IOTA directly supports this, e.g. the block lattice in NANO does not. Secondly, the transactions in IOTA are feeless (and only require Proof of Work to transmit), therefore no additional fees for the data storage occur. And, last but not least, this platform is already widely used and popular.

But we also need to discuss the drawbacks of this approach.

As the contracts live independently in their own chains, no direct interactions between them are possible. Instead of enforcing this strictly, we can define "namespaces" as grouping of dependable contracts (similar to Sharding). This means that CryptoPenguins, and every other App that depends on them, can be in the same namespace. Several approaches to get interoperability between different blockchains are currently researched, and those solution could be applied here as well. Also, asynchronous calls between different namespaces could be a worthwhile topic for future research.

This is a second-layer solution on top of the Tangle, and therefore unfortunately can not directly use IOTA as currency (because IOTA nodes do not understand the EVM protocol, and e.g. can not transfer any money out of smart contracts). Therefore an additional currency, limited to one particular chain, for fees and payment would be required. Methods for converting between IOTA and that token can exist, but the exact implementation for this token is out of scope for the Proof of Concept. Therefore our EVM-compatible prototype is feeless, and no *payable* methods are allowed. But, it should be noted, that this is only to simplify the Proof of Concept, this missing functionality could be added later.

6.1 Architecture

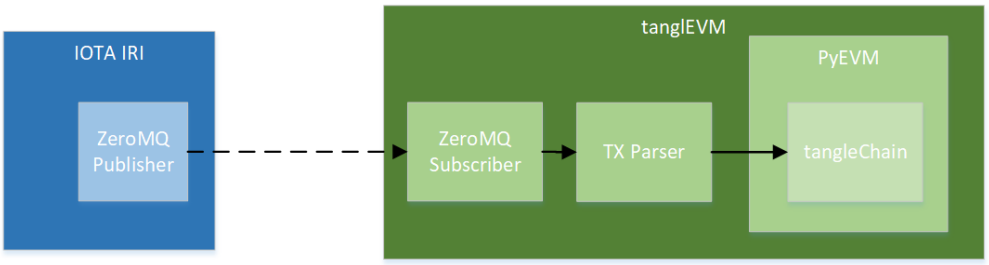

Figure 6.1: tanglEVM architecture overview

Figure 6.1 shows the architecture of our project.

There is an IRI daemon running, that is the equivalent of a full node in IOTA. Usually this node directly connects to the mainnet of IOTA. For testing purposes, the IRI can also provide a complete private testnet. This testnet also can run an instance of the IOTA Coordinator (*compass*), which issues IOTA milestones. This private testnet can be started easily in a container via Docker[38]. When the IRI is started with the 'zmq-enable-tcp' flag, it also acts as a publisher for a Zero message queue.[39] There are a variety of events published in this message queue, but the most important ones for our use case are the transactions that have been recently appended to the ledger, which are streamed in real-time.[40]

Our *tangleEVM* subscribes to these events, and check whether they are also a valid *tangleEVM* message. If they are, they are given to a customized *Ethereum virtual machine*, which then executes the transactions. This part is also responsible for issuing *EVM milestones*, which behave similar to Ethereum's blocks, for achieving consensus. How this happens exactly will be shown in the subsequent section.

6.2 Consensus

Another important aspect is how *consensus* is achieved.

We can draw parallels to the infamous IOTA Coordinator here. For our first Proof of Concept this is a centralized service, appropriately called *EVM coordinator*, which periodically issues milestones (like blocks in Ethereum that would be mined). These milestone blocks link to the transactions included in that particular block (which users previously attached arbitrary in the Tangle), and therefore determine the execution order of the transactions. The big advantage of our DAG is that there can be indefinitely many of those coordinators – so a huge traffic of transactions in one DApp doesnt influence any other, unrelated ones.

The transactions issued by the coordinator are illustrated in figure 6.2. Subsequent snapshots (blocks) are linked, and each block includes zero or more transactions.

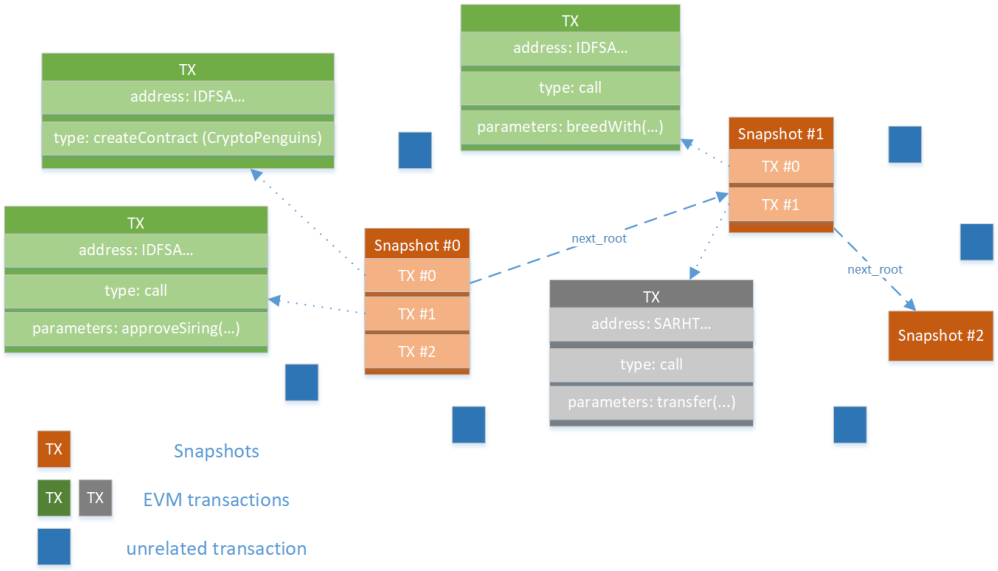

Figure 6.2: The tanglEVM Coordinator

While for now centralized, think of a Proof-of-Authority-type of network, solutions exist to achieve a more decentralized consensus, and can be implemented after further research (notice again the similarities to the Coordicide[28]).

6.3 Trytes

Table 6.1 highlights some of the most fundamental differences in the structures of the datatypes and algorithms used for hashing, and for representing addresses.

Property	Ethereum	IOTA
Address length	20 bytes	81 trytes[1]
Address representation	40 hex characters[2]	81 [A-Z9] characters
Address generation	Keccak-256 hash	Kerl hash[3]
Transaction Hash	32 byte Keccak-256 hash	81 trytes Curl-P-81 hash

[1] https://domschiener.gitbooks.io/iota-guide/content/chapter1/transactions-and-bundles.html
[2] excluding the '0x' prefix
[3] https://github.com/iotaledger/kerl

Table 6.1: Properties of Ethereum and IOTA

Section 4.2 establishes the concept of trytes, the fundamental representation of numbers in IOTA.

As the RLP-coded messages of the EVM are in bytes, we need to find some method to encode and decode between binary and trinary. Theoretically, there are $log(256)/log(3) = 5.0474...$ trits needed for each byte. The simplest method to encode a byte (with values in the range from 0 to 255) is to use a tryte which encompasses this range as well. This is possible by converting to a tryte made of 6 trits (a *tryte6*), which has a value range from 0 to 728 ($3^6 - 1$). The difference in the value range shows that there is quite a lot of wasted space, and consequently not every combination of *tryte6*s can be converted back to bytes. Figure 6.3 shows the conversions graphically.

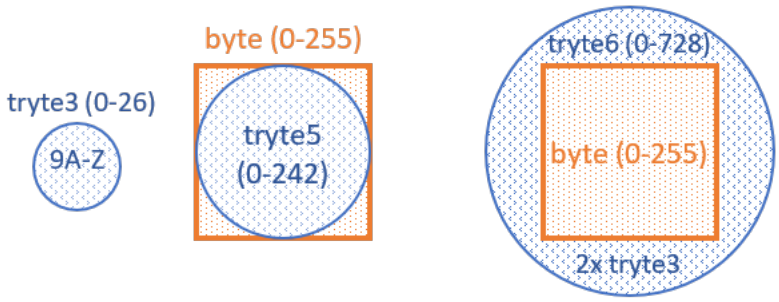

Figure 6.3: Tryte3 string, Encoding a tryte5 string as byte, Encoding a byte string as tryte6[82]

Based on the JavaScript library by vbakke[82], we use the Python code in listing 1 to encode a byte in a tryte6 string.

```
TRYTE_CHARS = '9ABCDEFGHIJKLMNOPQRSTUVWXYZ'

def encodeBytesAsTryteString(bytes):
  trytes = TryteString(b'')
  for value in bytes:
    value = int(value)
    firstValue = value % 27
    secondValue = int((value - firstValue) / 27)
    trytes += TRYTE_CHARS[firstValue] + TRYTE_CHARS[secondValue]
  return trytes
```

Listing 1: Encoding bytes as a tryte string

Conversely we can use the code in listing 2 to decode again a previously encoded string. Please note that this code will not work for arbitrary tryte strings, only for ones with values between *99* (0) and *LI* (255).

These methods allow the conversion from EVM bytecode, and any other RLP-encoded byte strings, to trytes.

```
def grouped(iterable, n):
  return zip(*[iter(iterable)]*n)

def decodeBytesFromTryteString(inputTrytes):
  if len(inputTrytes) % 2:
  return

  ba = bytearray(len(inputTrytes) % 2)

  for trytes in grouped(inputTrytes, 2):
    firstValue = TRYTE_CHARS.index(trytes[0])
    secondValue = TRYTE_CHARS.index(trytes[1])

    value = firstValue + secondValue * 27
    ba.append(value)

  return bytes(ba)
```

Listing 2: Decoding bytes from a tryte string

We acknowledge that converting the bytecode to a trytecode by changing all number representations, including the opcodes, would be a more efficient solution. This was not pursued, because a full compatibility of EVM bytecode and existing tools is desired.

While the proposed implementation is the most efficient solution that operates on single byte or tryte values, it might be feasible to operate on a grouping of byte or tryte values, and therefore losing less space. This would require added padding to the input strings though.[1].

6.4 TX Parser

As established in section 4.4 the used Winternitz one-time signature scheme reveals parts of the private key in the signature. This means that a private key, and consequently an address, can only be used once. This poses a problem for us, as we want to link all interactions of a user to a common identity. It would render the idea of smart contracts useless if a user could deposit money to a wallet contract, but would no longer be able to withdraw it because their address has changed.

Ethereum generates signatures via the *Elliptic Curve Digital Signature Algorithm*, which does not suffer from this problem. Although users can generate a multitude of accounts,

[1] see e.g. https://iota.stackexchange.com/questions/639/what-are-the-maths-to-convert-bytes-trytes for inspiration

interactions with smart contracts are always tied to a specific address, and following actions usually need to origin from the same account.

Therefore we need to find a way to tie multiple IOTA addresses to a single account. Value transactions solve this conundrum by ensuring that the whole balance of an address is spent with each bundle. The wallet creates a new, unused address (the *change* address) by incrementing the address index, and sends the remainder of the balance there. The spent address is no longer of any value, and should never be used again for a transaction.

Several methods already exist, which provide similar solutions for zero-value transactions. We will briefly present the most prominent ones, and discuss their usability for our PoC.

For our PoC we want to emulate an EVM-like message stream, therefore our main focus will be on the following criteria:

- **Message streams**. We want to send a multitude of EVM transactions from a single account.
- **Stream ID**. We want to uniquely identify the sender of a transaction.
- **Signatures**. Transactions need to be signed from the owner of the account, e.g. using the Winternitz one-time signature scheme.
- **Public access**. The messages should be publicly readable, and not encrypted in any form.
- **Index-based access**. Similar to the *nonce* in Ethereum transaction, direct lookup of a message just by stream ID and index would be desirable.
- **Python**. As our PoC is written in Python, an implementation of the mechanism in Python - or, some well-documented code in another language, which can fairly easily be ported - is desired.

6.4.1 Masked Authenticated Messaging

Masked Authenticated Messaging, or *MAM* of short, is an official IXI module (see 4.8) of the IOTA Foundation. This means it is a second-layer solution, which uses the Tangle for data access and storage. It acts as a data communication protocol, which allows clients to access communication channels, as well as methods to publish messages in such channels.[31]

Figure 6.4 shows the data streams of MAM in a high-level perspective. Every message in this stream holds a reference to the subsequent message. Readers of the stream, which are able to decrypt the messages, can find this connecting pointer in the attribute *nextRoot*. There is no method to trace back to the previous messages. If a reader is given the root of a message in the middle, only the subsequent messages can be read, therefore providing perfect forward secrecy. Obviously in the *Public* mode, the whole DAG could be scanned for a matching root, but this is impractical.

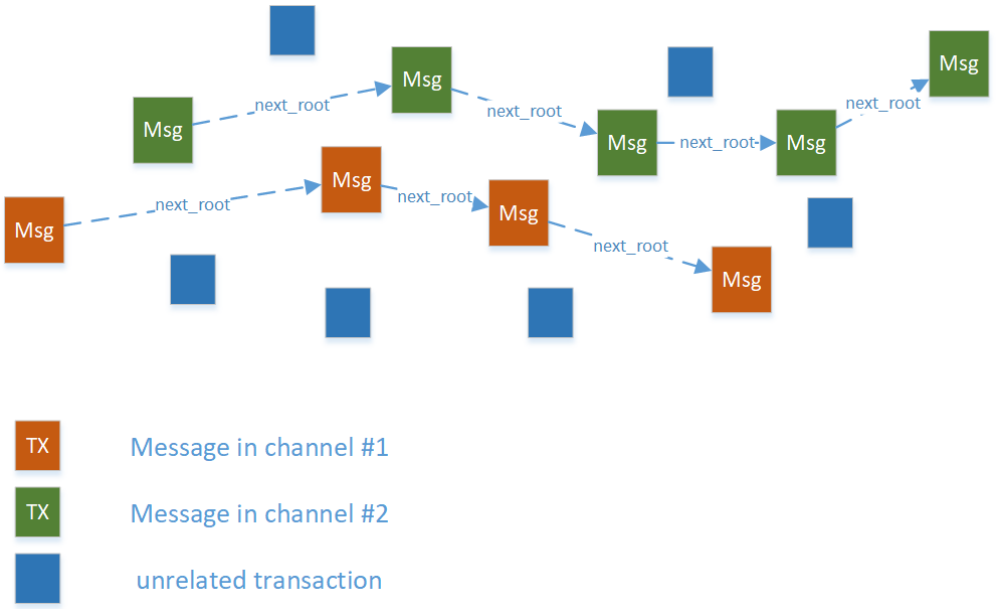

Figure 6.4: IOTA MAM architecture overview

Due to the usage of the Tangle, integrity of the data is ensured. It also fulfills privacy standards, by optionally encrypting the message stream. Three privacy modes exist, and they diverge in the way the channel ID is generated. In the *Public* mode everybody is able to read every message, the channel ID is simply the *merkle root*, and therefore everybody has the needed merkle root for decrypting the datastream by definition; in the *Private* mode the channel ID is the *hash of the merkle root*, and only users who know the merkle root can decrypt the message stream; in the *Restricted mode* the channel ID is also derived from the *hash of the merkle root*, but both the merkle root and another encryption key (sidekey) are required to be able to access the data stream.[2] For our further analysis, we only focus on the *Public* mode, because this is the desired access mode for our project.

Another feature of MAM is that channels can be forked into multiple streams (see figure 6.5). When using such a fork or splitting of channels, future messages use a new Merkle tree, whose root has never been revealed before. This can be used for fine grained access to specific subsets of the messages, with optionally stricter permissions. We will not provide further details of this feature, as it is not used in our application.

Figure 6.6 shows the Merkle tree based signature scheme. Trees can vary in size to accommodate for channel splitting, and this size can be different in every iteration. The size, that is the number of leaves and therefore channels splits for a given root, gets specified in the *count* parameter. A MAM transaction also holds a reference to the merkle root of the following MAM transaction in the *next_root*, which means that the next tree also needs to be generated in advance. The size of the next tree is specified in

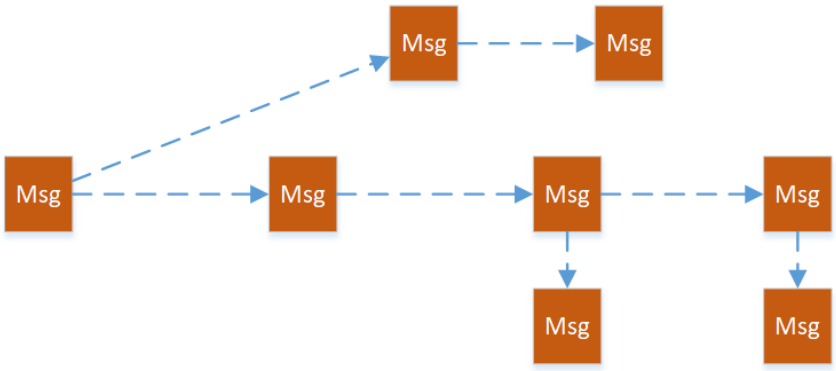

Figure 6.5: Channel Splitting in IOTA MAM

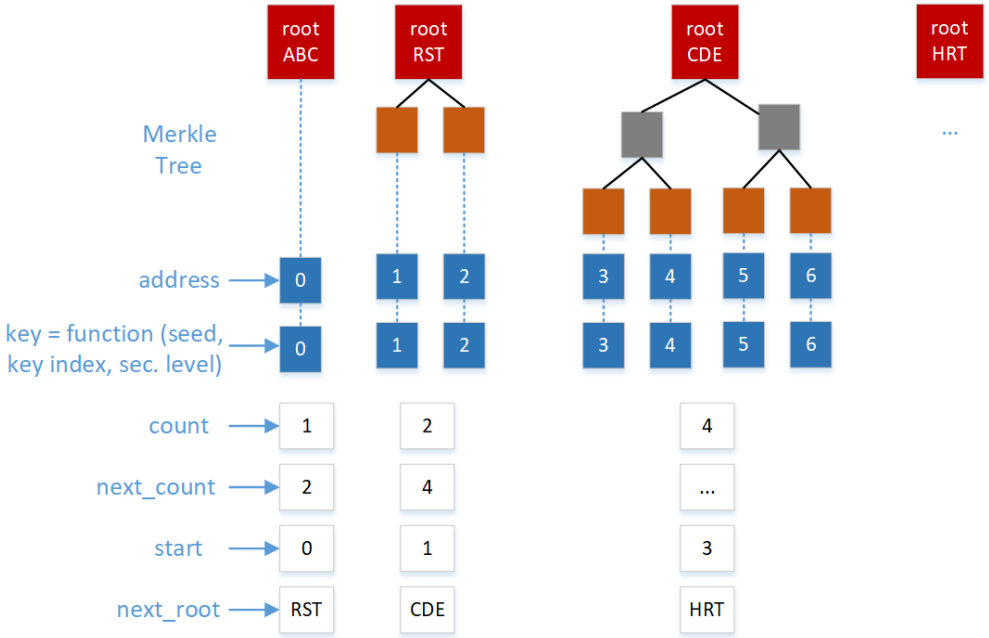

Figure 6.6: Merkle Trees in MAM used for signatures; source: mobilefish.com[49], IOTA tutorial 19 - Masked Authenticated Messaging

the *next_count* parameter. The *start* parameter refers to the first leaf key index number of the message chain. For the tree generation, also the mode, and the optional side key are needed, as these parameters influence the next root. The *security level* parameter specifies the size of the signature.

Not all of these fields are actually embedded in the MAM transaction, as can be seen in the structure of a transaction in figure 6.7. Besides the index, length, payload, next root, nonce, and signature, also parts of the merkle tree (the siblings) have to be included in a MAM transaction. This means that the overhead when utilizing MAM is quite large.

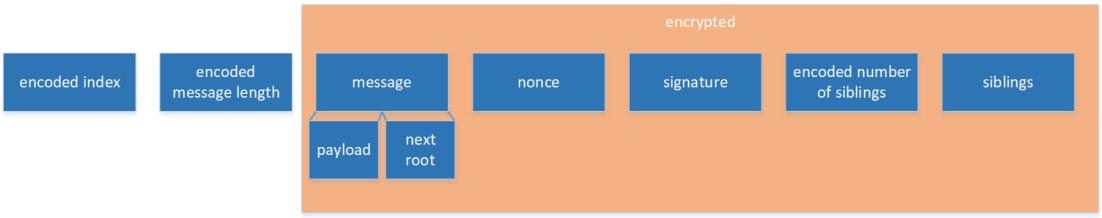

Figure 6.7: Fields of a MAM transaction; source: mobilefish.com[49], IOTA tutorial 20 - Masked Authenticated Messaging Payload

Even when the payload is completely empty, the Merkle tree has the minimal size of a single leaf, and security level 1 is used, the total size is already 2301 trytes, which does not fit into a single IOTA transaction (because the signature alone is 2187 trytes). With security level 2, the total minimal size is 4488 trytes, again bigger than 2 transactions.

Contract	method	bytecode length	transaction cost	trytecode length	MAM message length	# transactions
SuperToken	*deploy*	5386 bytes	1361546 gas	10772 trytes	15290 trytes	7
SuperToken	`transfer (0xAda.., 4914)`	68 bytes	51368 gas	136 trytes	4689 trytes	3

Table 6.2: Sizes for Transactions using MAM

In table 6.2 we summarized what this overhead means for a real life scenario. We used the *SuperToken* contract from section 3.5, and deployed it in our *tangleEVM* testnet chain. Using security level 2 for MAM, the deployment code of the 5386 bytes long contract is fragmented over 7 IOTA transactions, which can be deployed in slightly under 2 minutes in the Tangle (the Proof of Work for a single IOTA transaction takes around 16 seconds). Typical contract invocations take 3 IOTA transactions, which is the same as an IOTA token transfer (input, output, change transaction) takes. The conclusion here is that the overhead is not too bad, and such contract deployments and invocations are actually feasible.

MAM is implemented in Rust[2], and at the time only has client bindings in JavaScript[3], which are implemented via a JavaScript wrapper.

The project is still under development, and unfortunately detailed information or documentation does not exist yet.

[2] https://github.com/iotaledger/MAM
[3] https://github.com/iotaledger/mam.client.js

The following list summarizes the results of our research into MAM, whether the desired features are available and their viability for our project according to the needs we laid out previously:

- **Message streams**. Yes, this is the main feature of the project. MAM has the concept of *MAM Channels*, where owners of the channel can publish data, and other users can subscribe to this channel. This channel is secured by the Seed, so only the owner has the possibility to publish messages for a particular stream.
- **Stream ID**. Partly viable. The merkle root acts as a channel ID. But every message of this protocol links to the root of the next merkle tree, which provides forward secrecy. In our case this is bad, because we cannot get back to a previous message (and use the merkle root of the first message as ID). We also do not know whether a given message is the first one of the stream.
- **Signatures**. Yes, messages are signed using the Winternitz one-time signature scheme.
- **Public access**. Yes, a stream can be public, private or restricted.
- **Index-based access**. No, only linear access of subsequent messages.
- **Python**. No, the only currently existing implementation is the reference implementation in JavaScript, which accesses a library written in Rust. Adding support for Python is not planned for the near future[4].

6.4.2 MAM Lite

MAM Lite is basically the same as *MAM*, but utilizes the RSA signature scheme to sign and verify messages, instead of the W-OTS as MAM uses.[5]

As it generally offers the same features as *MAM* (private, restricted and signatures modes, channel splitting, forward secrecy) no in-depth research into this project was done by us.

No development has happened since November 2018, which leads us to conclude that it is currently no longer maintained.

6.4.3 MAM Ultra Lite

MAM Ultra Lite is another project inspired by both *MAM* and *MAM Lite*, that again changes the signature scheme.[6]

[4] https://github.com/iotaledger/iota.lib.py/issues/47

[5] https://medium.com/@samuel.rufinatscha/mam-lite-a-more-flexible-messaging-protocol-for-iota-562fdd318e1d

[6] https://medium.com/coinmonks/iota-mam-ultra-lite-493d8d1fb71a

They claim that RSA signatures need too much space, and therefore use Ed25519 signatures, the same scheme as the cryptocurrency NANO (section 1.3) uses. Compared to the 1128 trytes for the public key and 1024 trytes for the signature used in RSA, this is certainly the case with only 86 trytes for the public key and 172 trytes for the signature in Ed25519. As Ed25519 cannot be used for encryption, private access still needs some additional RSA encryption added, but as this is not needed for our project, it can be ignored.

The big advantage of using this method would be that the implementation is directly written in Python, and therefore would integrate easily into our project. Besides that, there are no clear additional benefits, and therefore also no in-depth research was performed.

6.4.4 Random Access Authenticated Messaging

Random Access Authenticated Messaging, or *RAAM* for short, is another module for message streams in IOTA's Tangle developed by Robin Lamberti[45] and published as open-source library[17].

While *RAAM* is also inspired by the *MAM* module, it finally deviates from the design of *MAM* in more substantial ways than just in the algorithm used for signatures.

The big advantage here is the indexed access of messages in a channel, in a time of $O(1)$. No chain, beginning from the very first message of a channel, needs to be followed to be able to access any arbitrary index. Figure 6.8 shows that a message address can be calculated, just from knowing its channel ID and index.[45][17]

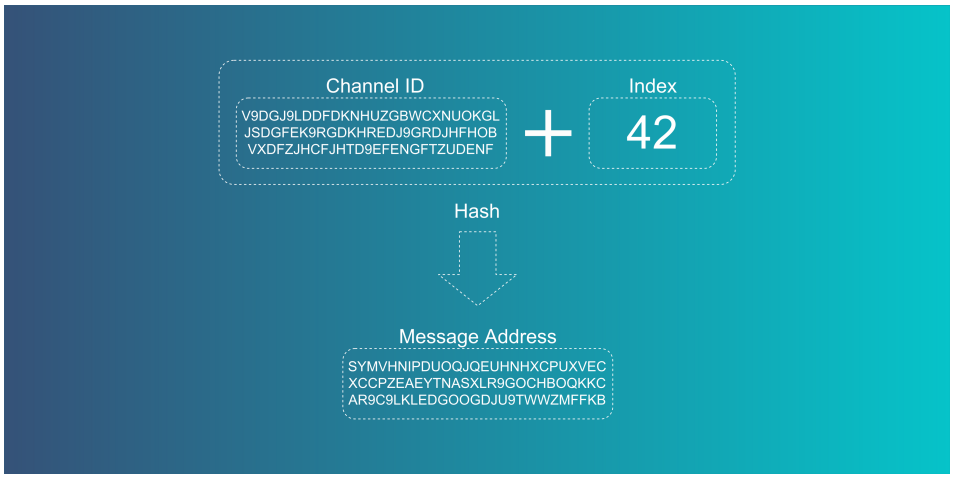

Figure 6.8: The computation of a message's address[45]

RAAM also uses Merkle trees to cryptographically proof that someone who holds the private signing key for a message also holds all other keys used in the channel. While in MAM every message contains its own (small) merkle tree, in RAAM the complete tree

of all possible signing keys is generated upfront. This approach is similar to how the IOTA coordinator (section 4.7) generates its signing tree.

Depth	# transactions	Time for security 1	Time for security 2
8	256	5.2s[1]	8.2s[1]
9	512	8.3s[1]	14.8s[1]
10	1024	15.9s[1]	29.9s[1]
11	2048	28.5s[1]	53.2s[1]
12	4096	1m 0.5s	1m 48s
13	8192	2m 0.4s	3m 54s
14	16384	3m 42s	7m 46s
15	32768	7m 36s	14m 43s
16	65536	15m 9s	28m 5s

[1] Average value of three runs

Table 6.3: Merkle Tree generation time, depending on the layer size and security level

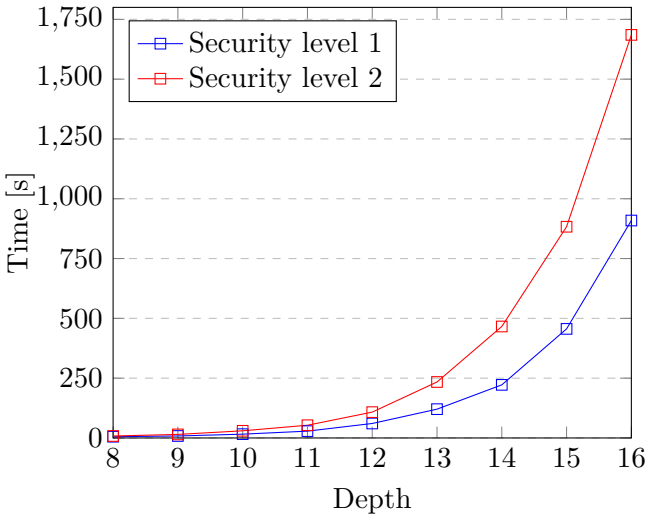

Figure 6.9: Merkle Tree generation time, depending on the layer size and security level

To calculate the feasibility of such a pre-computation of a Merkle Tree we ran some tests to find out how long the generation takes [7]. The results on our hardware (Intel Core i7-3770 CPU @ 3.40GHz, 16GB RAM) are shown in table 6.3 and figure 6.9. The command used to generate this data was:

[7] for comparison, the IOTA coordinator has a depth of 23, which can generate 8,388,608 public/private key pairs

```
time docker run -t -rm -v $(pwd)/data:/data iota/compass/docker:layers_calculator
layers_calculator_deploy.jar -sigMode CURLP27 -seed $seed -depth $depth -security
$security -layers /data/layers
```

Initially *RAAM* only had a private mode, where readers needed to know the channel ID (the merkle root of the merkle tree) to be able to access the stream. This channel ID was not encoded into the message, so no public access was possible.[8] After a conversation with Robin he was kind enough to think about adding support for a public, non-encrypted mode. After some time the project really did gain this feature by adding the concept of public messages, which can be decoded solely with their respective address. This is done by directly encoding the channel ID and the index into the message.[9]

Contract	method	bytecode length	transaction cost	trytecode length	MAM message length	# transactions
SuperToken	*deploy*	5386 bytes	1361546 gas	10772 trytes	17496 trytes	8
SuperToken	`transfer (0xAda.., 4914)`	68 bytes	51368 gas	136 trytes	6561 trytes	3

Table 6.4: Sizes for Transactions using RAAM

We also performed tests encoding the EVM deployment message for the *SuperToken* contract and an EVM message calling the *transfer* function with *RAAM*. The results can be found in table 6.4. To make it comparable with our results for *MAM*, we again used security level 2 - with level 1 the messages could be smaller, and the contract call would fit into 2 IOTA transactions.

The results of our research into this library are:

- **Message streams**. Yes, this is the main feature of the project.
- **Stream ID**. Yes, there exists a global channel ID.
- **Signatures**. Yes, messages are signed using the Winternitz one-time signature scheme.
- **Public access**. Yes, this feature was recently added and now it is possible to get the channel ID just from knowing an address.
- **Index-based access**. Yes.

[8] also the message at index 0 does not match the channel ID, but is just the address of first leaf in the tree

[9] https://github.com/cr0ssing/raam.client.js/commit/75bfeaa46ad9e2a6be2c9e87bcdb4a2a4c132e4c

- **Python**. No, not at the moment, but planned. The project is also fairly simple, a port to Python would therefore not take much time.

6.4.5 Discussion

We have chosen to use *MAM* for our PoC, because it was and still is the most commonly used message stream implementation for the Tangle, and *RAAM* was still lacking the public mode support during its development. But this decision can be re-evaluated in future, as the static channel ID and the index-based access of *RAAM* better suit our needs for the *tanglEVM*.

It has to be noted that all non-value transactions in the Tangle can get removed by nodes after a snapshot of the network is performed[10]. Such snapshots are regularly performed, wiping the complete non-value transfer history of the Tangle. Therefore users need to connect to permanodes to be able to get the complete history.

Another confirmation that this idea is on the right track can be seen by a blogpost by the IOTA foundation on November 26th 2019. When it was published, this thesis was still in work. In the article "Integrate Hyperledger Fabric with the IOTA Tangle"[74] IOTA developers describe how smart contracts written in the *chaincode* of *Hyperledger Fabric* can be integrated into the Tangle by storing the results of smart contract executions. The connection to IOTA is formed here as well via *MAM message streams*, further showing the potential of our proposal. They are satisfied with the results they achieved so far, and are exploring further options to expand these capabilities. This shows that the Tangle is a capable storage medium for the interactions of different smart contract platforms.

6.5 Py-EVM

For our state and computation machine we make use of the reference implementation of the EVM in Python, an open-source project called Py-EVM[11]. Due to its modular architecture it is highly flexible, and is intended to be used both as a base layer for *full* and *light* Ethereum clients, as well as a research platform for future features and alternative use cases like private chains. This modular architecture is of great benefit for our project, as it is relatively easy to change certain aspects of the EVM.

6.5.1 Architecture

Figure 6.10 shows a high-level overview of the Py-EVM architecture. It is split into five major components: the *VM*, that handles the protocol aspects of the EVM; the *Chain*, that initializes the parameter for common use cases like the mainnet or testnets; a *Storage*

[10] Recent versions of the IRI added the possibility to perform local snapshots by the nodes themselves, so the actual history can differ from node to node[10].

[11] https://github.com/ethereum/py-evm

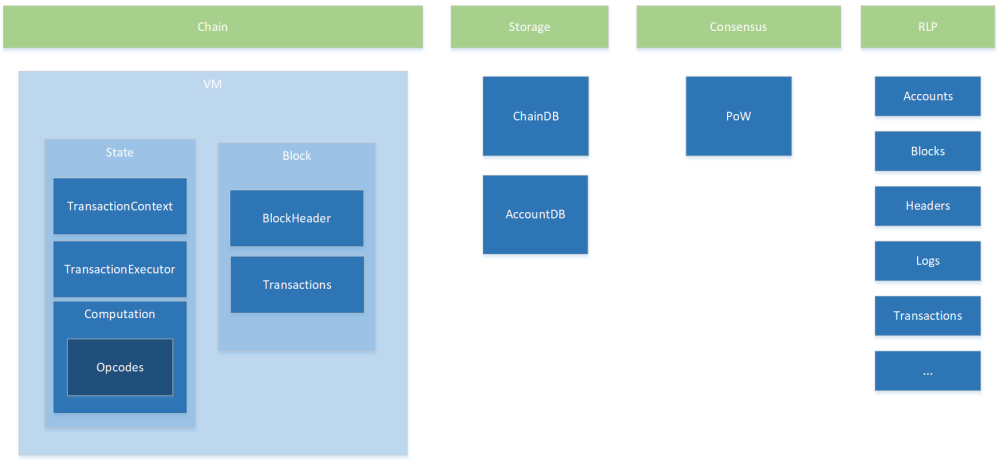

Figure 6.10: Py-EVM Architecture Overview

layer that implements how objects are cached and stored in various database backends; the *Consensus* module which deals with calculating (although not recommended for mining on the mainnet, as it is implemented purely in Python) and verifying the Proof-of-Work algorithm, and finally the *RLP* helper, which acts as helper to specify how the objects get serialized and deserialized in RLP encoding.

6.5.2 Chain

The **Chain** is basically an orchestration layer. It defines at which block (e.g. starting from the genesis block) which implementation of the Virtual Machine (e.g. the *FrontierVM*) should be active. It further defines which storage implementation should be used for the chain. All other calls are then passed through to the currently active VM, or to the database backend.

Listing 3 shows a blockchain, which uses the Frontier VM for the first 1150000 blocks, and then switches to the Homestead VM, like it does in the Ethereum mainnet.

In our case, we do not need to implement the legacy VMs, so we can configure that it should directly activate our custom VM implementation right from the beginning (the genesis block).

```
from eth import constants, Chain
from eth.vm.forks.frontier import FrontierVM
from eth.vm.forks.homestead import HomesteadVM
from eth.chains.mainnet import HOMESTEAD_MAINNET_BLOCK

chain_class = Chain.configure(
  __name__='Test Chain',
  vm_configuration=(
    (constants.GENESIS_BLOCK_NUMBER, FrontierVM),
    (HOMESTEAD_MAINNET_BLOCK, HomesteadVM),
  ),
)
```

Listing 3: Chain configuration for Ethereum mainnet with Frontier and Homestead VM implementation

6.5.3 VM

To ensure compatibility with EVM bytecode our implementation is based on the Petersburg Fork (figure 6.11), so all existing smart contracts are also able to run in our *tanglEVM*[12].

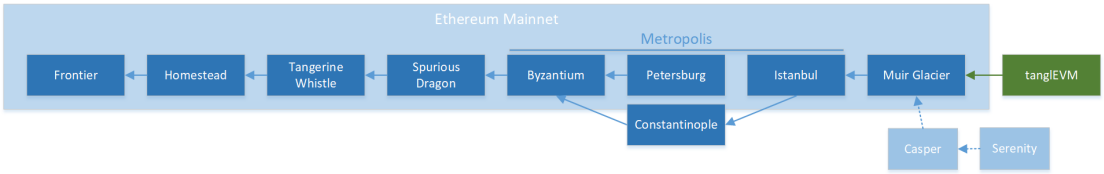

Figure 6.11: tanglEVM in relation to Ethereum forks

Since we are using a customized VM implementation, we are also free to change the behavior of the VM. This can be used to better integrate the blockchain-based EVM with our DAG ledger. The aim is to loosen the strict segregation of our introduced namespaces, so function calls between different namespaces are possible again. To answer the important question of what exactly needs to be adjusted to achieve this, another literature study was performed first.

In this literature study, we stumbled upon the Vite project, which offered a few promising ideas.

[12]It is important that every class inherits the traits from the desired fork. If, for example, the computation engine would be based on the older *Byzantium* fork, execution of contracts compiled with recent compiler releases might fail. A *SuperToken* contract might then fail with the error message "*eth.exceptions.InvalidInstruction: Invalid opcode 0x1b*", because the opcode for *Shift Left* was only implemented in the succeeding fork of the EVM. This might cause some confusion.

Vite

We have analyzed a cryptocurrency project that tries to achieve something similar to our project. Their goal is to create a DAG-based ledger, with support for EVM-like smart contracts, which basically aligns with the topic of this thesis as well.[50]

This project also has a smart contract platform, that tries to stay as compatible with the EVM as possible, while still changing certain properties. The whitepaper only mentions that the smart contract language *Solidity++* is similar to Ethereum's *Solidity*. Also the opcodes themselves resemble the ones in the EVM - e.g. while the hashing function *Keccak* was changed to *Blake3*, the transfer and balance function take an additional parameter to natively support tokens. Several opcodes (TOKENID, RANDOM, etc.) have been added, and the contract call logic is different to be able to implement asynchronous calls. For the rest of them no big difference to the implementation in the EVM could be found.

We used this approach as guide for our project. It is of particular interest to us how they changed the semantics of message calls to an event-driven architecture. Their scalability concerns lead to a change of the *ACID* (Atomicity, Consistency, Isolation, Durability) scheme of the EVM to a final consistency scheme *BASE* (Basically Available, Soft state, Eventual consistency). This leads to a design, where function calls are no longer synchronous across contracts, but rather an asynchronous message communication. The EVM instructions *CALL* and *STATICCALL* no longer immediately execute the called contract, and therefore also no longer return the result of the call[13].

The consensus mechanism also works in an asynchronous way by splitting the transaction into *request* and *response* pairs, allowing the writing of transactions into the DAG ledger without being blocked by the confirmation process.

While the basic principles of this project inspired the design of our prototype, we argue that developing yet another independent ledger might not be the best approach, and piggybacking the smart contract support on an already existing, well-established DAG ledger is a worthwhile solution.

We also do like the asynchronous message communication of *Vite*, but their implementation in the smart contract programming language *Solidity++* does not look quite elegant enough. We would prefer Python-like *async/await* syntax for these asynchronous calls.

The semantics of most Vite instructions are equivalent to the EVM. But while being similar, it is a complete reimplementation of the EVM, and no actual source code is shared between the two systems. Our design directly inherits an official implementation from the EVM, and it is therefore easier to keep up with changes in the EVM, as they do not need to be rewritten for our implementation.

[13] the result of these instructions is hardcoded to *0*

Opcode	Mnemonic	Stack Input	Stack Output	Expression	Notes
e0	CURL	offset | length	hash[:18] | hash[18:]	$hash = curl($ $memory[offset : offset + length])$	hash split into two parts to fit size

Table 6.5: Opcode *Curl*

Extending the EVM

The previous section made clear that changes to the semantics of several EVM instructions are desired. We just do not agree that these changes should replace the mechanisms of the EVM, they rather should exist as their own opcodes. Before implementing them in detail, an understanding of how difficult changes to the implementation EVM opcodes generally are was required.

For testing our assumption, we decided to implement another opcode. A simple candidate for this was the hashing function *Curl*, which is supported natively by *IOTA*, but does not exist in the EVM. The semantics of it are shown in table 6.5.

The source code of the implementation for this opcode can be found in listing 6. Here we can see that the EVM only operates on bytes, therefore the tryte input has to be decoded first. Then the operation can be performed, and finally the result has to be encoded again as EVM-compatible bytes. Such tedious conversions would not be practical for real-world scenarios.

The input on the stack is the same as for the similar *Keccak256* operation, it takes an offset in the memory region, where the to-be hashed data is lying, and the length of the data. But for *Keccak256* the output - the hash result - fits nicely onto the top of the stack as an individual item because this matches the 256 bits word size of the EVM. But the result of a standard Curl-P-81 is 81 trytes (or 243 trits) long. Even worse, it has to be encoded again as bytes, making it substantially bigger than the word size. Therefore the result needs to be split into two values (which does waste some space at the end).

Arguably even more difficult than implementing a new opcode is constructing a smart contract that uses it. Until now we only instructed the computation engine of the EVM what it should do when it encounters the *curl* opcode, but not the *solc* compiler how it can generate bytecode that calls this instruction. Patching the compiler turned out to be a work too enormous for this thesis, so we looked for other ways to test whether our implementation is working correctly. Solidity has some support for inline assembly in the language[14], but this syntax also only supports already known mnemonics, not a brand-new one like implemented here.

The only viable possibility therefore was to write bytecode ourselves. We started out with a similar contract in Solidity (see listing 4), which has a function that simply hashes its arguments (via *keccak256*, which will be changed later by hand), and returns the hash result (already as two *bytes32* values, as the extra space will be needed afterwards).

[14] https://solidity.readthedocs.io/en/latest/assembly.html

```
pragma solidity ^0.5.0;

contract Hash {
  function myhash(bytes memory data) public pure
           returns (bytes32, bytes32)  {
    bytes32 a = keccak256(data);
    bytes32 b = "";
    return (a, b);
  }
}
```

Listing 4: Source code of *Hash* contract

We used the *solc* compiler to compile this contract, with extra options to generate a listing of opcodes, and an EVM assembly as well: `solc -opcodes -asm -bin -abi -o Hash Hash.sol`

Let's walk through the process step by step:

The runtime segment of the assembly is presented graphically in the call graph of listing 6.12.

We now need to understand what exactly happens on a bytecode level. The blog series *Deconstructing a Solidity Contract* by *OpenZeppelin*[70] provides a good introduction at demystifying the EVM bytecode produced by the Solidity compiler, and we will follow a similar approach. Following their how-to is still encouraged, as it provides a further in-depth analysis.

We can safely ignore the creation code (the whole Part II of the *OpenZeppelin* manual), as this is the same for every contract creation, and focus solely on the runtime code. And also the beginning of the runtime is mostly just boilerplate code - the free memory pointer gets defined and a calldata length check is performed. The calldata is an encoded chunk of hexadecimal numbers that contains information about what function of the contract we want to call, and its arguments or data. Then the function dispatching happens: function signatures (introduced in section 3.5.5) are checked whether they match, and if they do redirect the program flow to a function wrapper.

The inner workings of the function wrapper are also not important for us. It is sufficient to know, that it encloses the function body and provides the environment for the function body by unpacking the calldata, routes execution to the function body and then repacks whatever comes back for the user (in the *memory returned structure* of the graph).

The function body, in the orange block of the call graph, is where the actual hashing function is called. For convenience, the actual opcodes are visible for the assembly as well. Upon entering this part, our stack looks like this:

80 *memory pointer*	e9 *return address*	7dc4a9ec *calldata*

At the beginning, space for the later returned values is reserved:

00 *bytes32*	00 *bytes32*	80 *memory pointer*	e9 *return address*	7dc4a9ec *calldata*

The next few lines prepare the arguments for calling the *keccak256* opcode on the stack - it needs the memory offset of the to-be hashed data (which is at offset $0x20$ of the *memory pointer*, and the length of the data (see table 3.2 for a description). Right before the instruction, the stack contains the following elements:

a0 *offset*	01 *length*	00 *bytes32 a*	00 *bytes32*	00 *bytes32*	80 *memory pointer*	e9 *return address*	7dc4a9ec *calldata*

After the execution, the parameters are removed, and the result is the top-most element on the stack:

ec97e60... *keccak256(data)*	00 *bytes32 a*	00 *bytes32*	00 *bytes32*	80 *memory pointer*	e9 *return address*	7dc4a9ec *calldata*

Then some swapping is performed to store this value in the *bytes32 a* slot. We can see that the generated bytecode contains superfluous instructions in that part - running the compiler with optimization enabled would have cleaned up the code a bit (but would also have gotten rid of our reservation for the second *bytes32* value).

Finally, at the end of our function body, the stack contains the address of where to jump next (the *memory returner structure*, which ends with the *RETURN* instruction), and the result of our computation:

e9 *return address*	00 *bytes32 b*	ec97e60... *bytes32 a*	7dc4a9ec *calldata*

We can now modify the function body by hand to use our implemented *curl* opcode. As this opcode produces two values on top of the stack, we also get rid of the hardcoded `0x00` value on top of the stack we had to use as placeholder before. The changes to the body - highlighted in bold - can be seen in figure 6.13.

This now finally allows us to execute a contract call that uses our instructions to verify that it indeed works as specified:

$myhash(encodeTryteStringAsBytes(\texttt{'THIS9IS9A9TEST'}) =$
`0x000...0206187e79e0f...`$|decodeTryteStringFromBytes =$
`EJEAOOZYSAWFPZQESYDHZCG...`

We acknowledge that this opcode might not be the best candidate to implement in a fork of the EVM. This should only be seen for exemplary purposes, to get a feeling of how much work is required to extend the EVM. And similar methods can be used to actually do something useful, like implementing asynchronous calls across namespaces, or providing mappings between IOTA addresses (or channel IDs) and EVM addresses.

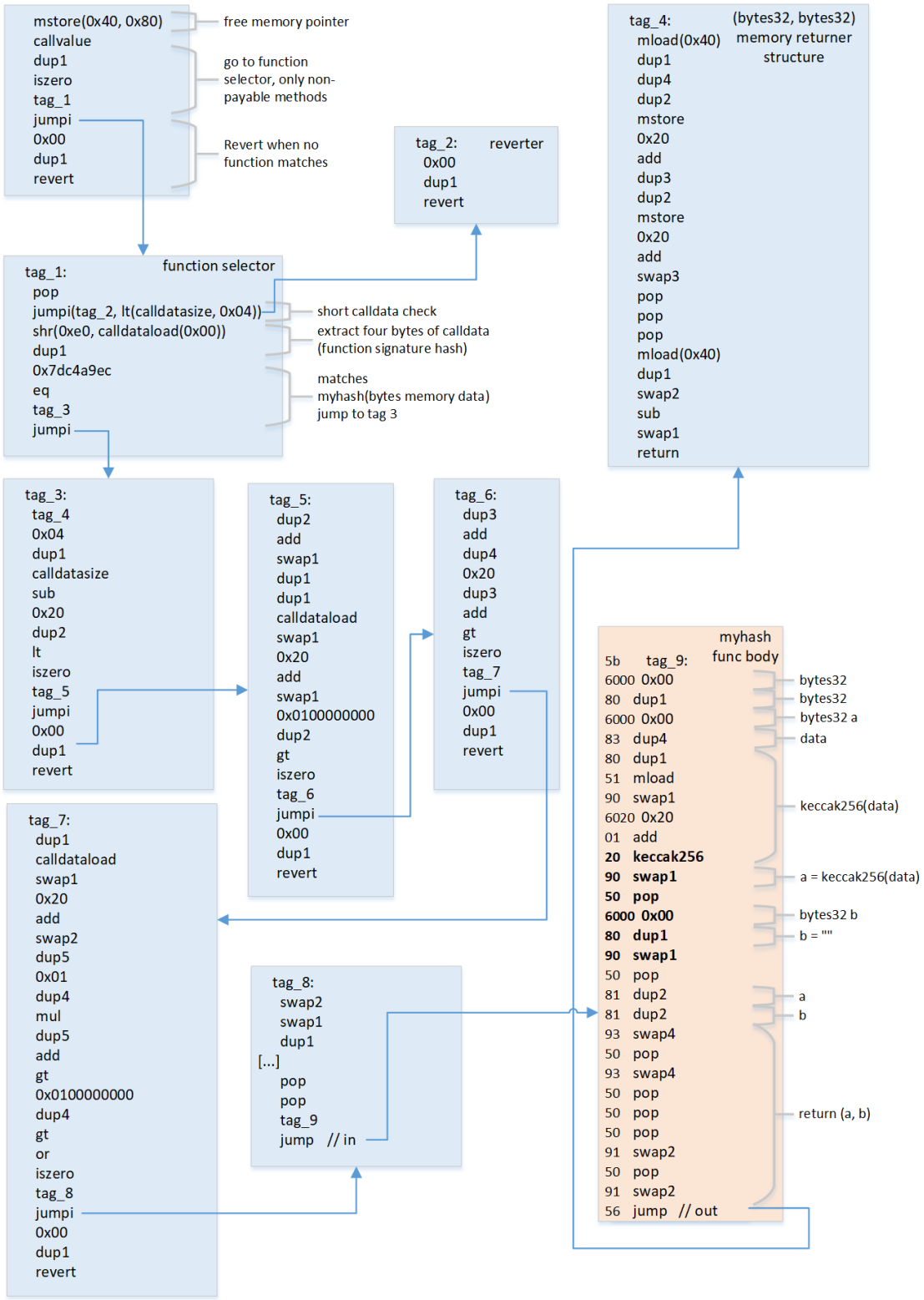

Figure 6.12: Call graph for Hash contract, implementing a *keccak* hash

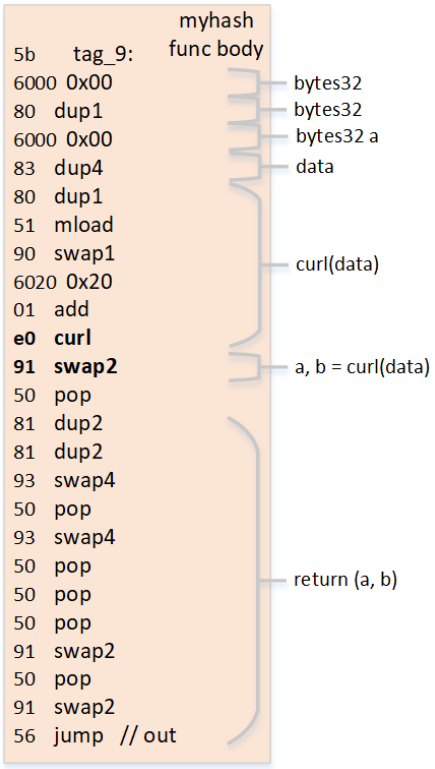

Figure 6.13: Manually modified *myhash* function body, to use *curl* hashing function

6.5.4 Database

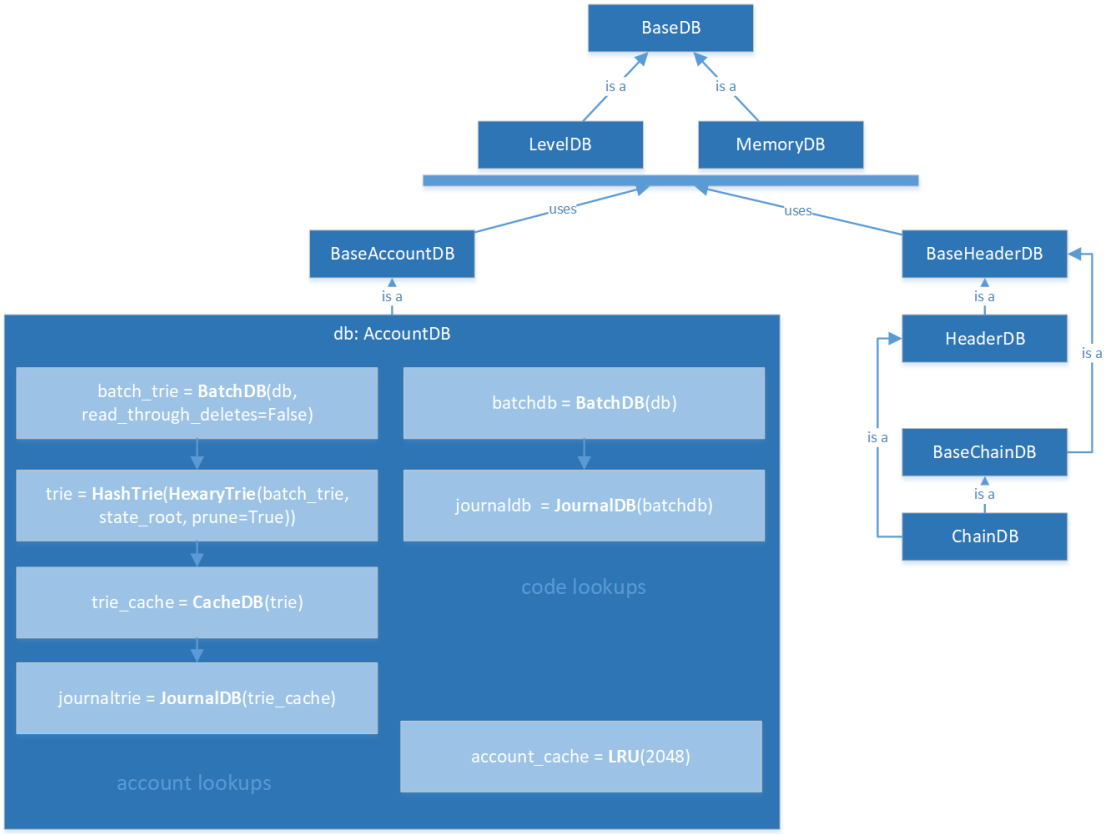

Figure 6.14: Database architecture of Py-EVM

Figure 6.14 shows the complex database layers of the Py-EVM. As physical storage backend two options are provided: either the data is only stored temporary in memory (that is a simple Python *directory*) and lost whenever the Py-EVM is terminated, or it gets permanently written to disk in a *LevelDB* database. *LevelDB* is a key-value storage library, which provides an ordered mapping from string keys to string values, and is developed for efficiency. Its API behaves similar to a Python *dictionary* object. Changes to the data can be made in atomic batches, this is a useful feature needed by the Py-EVM. The written data is automatically compressed, resulting in disk space savings when the huge Ethereum blockchain is synced.

The LevelDB keys are mostly *Keccak256*-hashes of the objects they represent (like the hash of the account address), and the value is the RLP-encoded bytestream of that object (like the RLP encoding of account fields). This means that, when just looking at the database, it is difficult to tell what exactly the content of a row represents, because neither key nor value contain any concrete type information.

The storage is split into two categories: the chain data, with Headers, Blocks, Transac-

Key	Value
v1:canonical_head_hash	\xa0\x80\xb7\x82\xcc...
block-number-to-hash:0	\xa0]\xe3S\x92...
block-number-to-hash:1	\xa0\x80\xb7\x82\xcc...
block-hash-to-score:]\xe3S\x92...	\x01
U\$U\xb0l\x10\x85...	\xf9\x01\xf7\xa0]\xe3S\x92...
\xfe\x88\x9f\x10\xe5...	\xf8q\xa03\x80...
\xfd.Y;\x89\xfb\xf9...	\xf8q\x80\xa0/...
\x87\x8a\xc5S\xb5...	\xf8Q\xa0(\xe7...

Table 6.6: Excerpt of a LevelDB database for a private Ethereum network

tions and Receipts; and the state data with balance, nonce, code and storage.

In table 6.6 we can see that the database also contains some organizational fields for the block header database, that make index-based lookups easier. The block hash of any arbitrary block index can be found at key *block-number-to-hash:N*, where *N* is the block number. Reverse queries are possible as well, prefixed with the key *block-hash-to-score*.

Following the block hashes, the RLP-encoded blocks can be found, which then can be followed to transactions and accounts by their respective hashes.

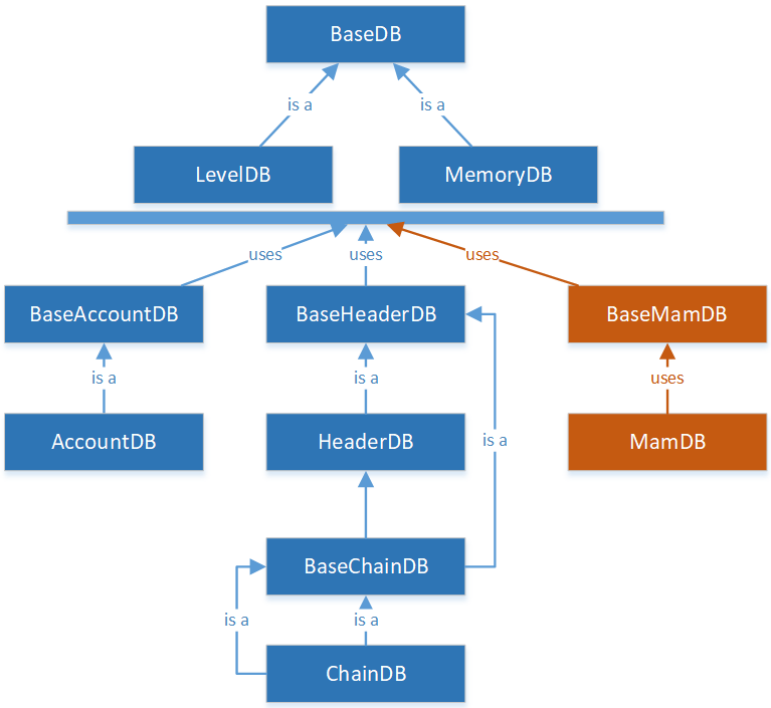

Figure 6.15: Py-EVM

We had to extend this database scheme, because our *MAM* abstraction layer also needed to store some metadata. Because *MAM* only allows easy access to subsequent messages of the stream, we need to store the *next root* field for every message of the stream in the database, alongside the *current root* and the *first root*, which acts as channel ID. When we encounter a new published message, we can then look up in $O(1)$ to which channel that message belongs. See figure 6.15 for a schematic of our changes.

Initially we wanted to extend the *Account* attributes with these fields, but that would have required substantial changes to the caching logic, and to other unrelated classes that make use of *Accounts*. For simplicity, we therefore moved the logic for assembling the message stream and parsing the channel ID to the *TX Parser*, and only give EVM-compatible messages to our *Py-EVM* implementation.

6.6 Future Work

We have deliberately omitted consensus methods, as this would go beyond the scope of this thesis. But with our proposal, each chain in the DAG could theoretically use their own appropriate method for achieving consensus. Selecting the right mechanism would be in the responsibility of the *DApp* developer, as the security requirements for each *Dapp* differ. For authoritative apps (like centralized exchanges), a proof-of-authority system (similar to the one proposed in this chapter) could be perfectly fine, while decentralized games (think of Cryptokitties) might want to use a *Delegated proof-of-stake* consensus method. This would even be fairly easily achievable, as each token can be seen as native currency in that specific chain. This not only allows trading of that currency, but also staking it to vote for a delegate.

Inspiration for the selection of a suitable method can be found in the paper by Burg[7], which lists a variety of algorithms.

6.7 Summary

At the end, we want to show the total picture that covers the topics of the whole thesis.

Figure 6.16 shows the contract invocations of our fictional user *Becca*. At the innermost layer is the RLP-encoded (sec. 3.6) EVM transaction in which she invokes the *transfer* method of the *SuperToken* contract to send 4914 tokens to the user Ada (see 3.5.5).

Due to the quirks of IOTA, this bytecode has to be encoded to a trytestring (sec. 6.3). In order to build a message stream of Becca's transactions, this is passed as payload to a public MAM message (see 6.4.1). The MAM message also holds a reference to the address, that her seed generates as next root of the next Merkle Tree. As usual, this next tree will also only contain a single leaf, meaning that no MAM channel split can occur.

Now this message of the MAM stream should be published in the Tangle. As its length exceeds the maximum permitted length for a single IOTA transaction, it has to be

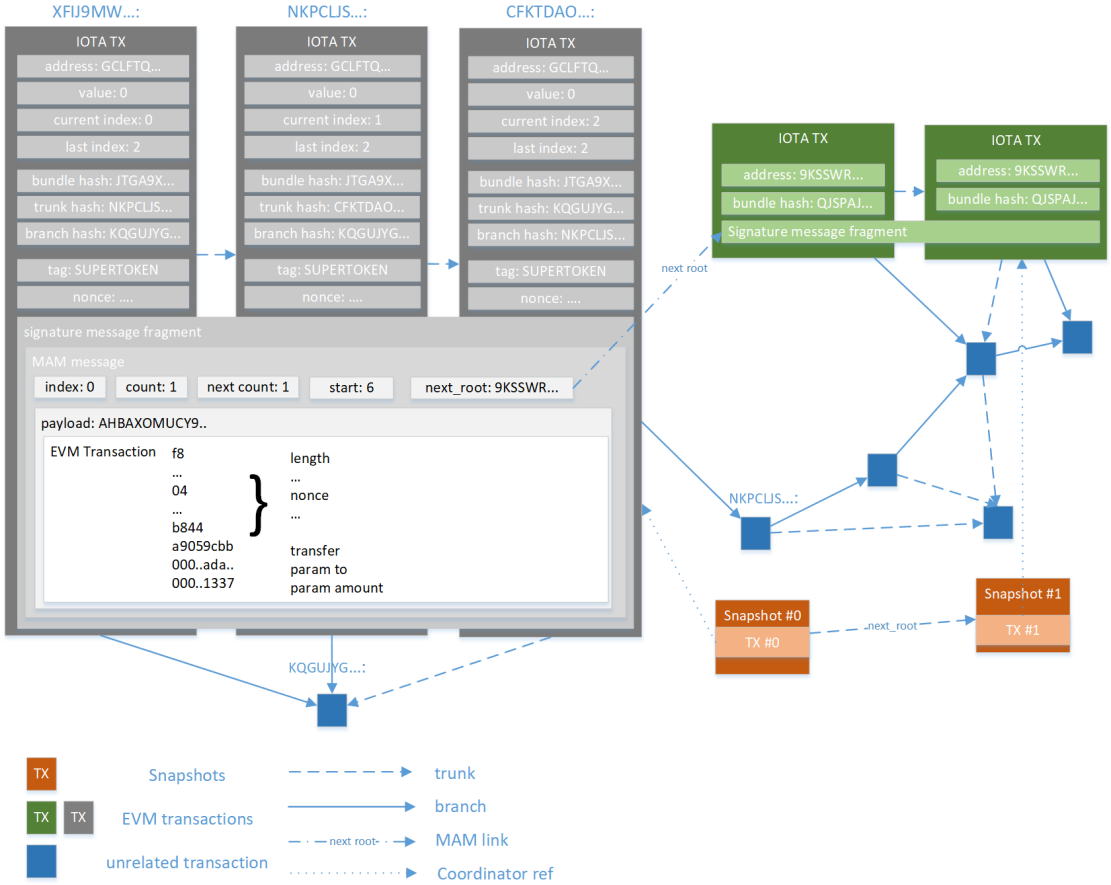

Figure 6.16: Total picture of a tanglEVM transaction

fragmented over multiple transactions, logically grouped via a bundle (sec. 4.5). Each transaction in this bundle has the same *bundle hash*, which is calculated via the "essence parts" (specific fields) of the bundle.

Each IOTA transaction is a zero-value transaction, because the *signature message fragment* contains (parts of) our MAM transaction, instead of a Winternitz signature (sec. 4.4) to prove the ownership of an address. As no signature validation happens at all, we can use arbitrary addresses (sec. 4.3). The specification for MAM is to use the *channel ID*, that is the Merkle Root, as address. To fetch future messages of this channel, it is sufficient to listen to this channel (and then validating the fetched message, as everybody can publish a message to that address).

For a bundle, IOTA also clearly specifies how the links to the two parent transactions, the trunk and branch, that a transaction validates, have to be set.

And finally, we show our centralized Coordinator, similar to the coordinator of IOTA itself (sec. 4.7), that periodically issues snapshot blocks, and is the basis for forming

consensus (sec. 6.2) in our DApp chain.

This simple method for achieving consensus puts itself forward to be decentralized in a future work (sec. 6.6).

CHAPTER 7

Conclusion

At first we have performed a survey of the existing smart contract distributed ledger platforms. While currently much development is happening in that space, *Ethereum* with its *Ethereum Virtual Machine* is still the most-used platform. Neo seems to be a viable contender, that focuses on aspects like smart economy and digital identity, but its smart contract engine *NeoVM* has no noticeable improvements compared to the EVM. We also noticed that most smart contract platforms currently developed tend to seek compatibility with the EVM, making the EVM the de-facto standard in this space. The EVM itself is still in development (and platforms trying to be compatible with it are having a hard time catching up), with a planned shift from smart contract-specific languages like *Solidity* to general purpose programming languages developed on top of the open WebAssembly (Wasm) standard.

We also found that it is not the smart contract virtual machine that differentiates the projects. Even in the rare cases when the VM is not directly based on the *EVM*, its functionality is identical in large parts: some stack-based execution engine implementing a state machine, that adds to the usual logical and arithmetic opcodes a handful more for cryptographic operations and querying the state of the blockchain.

It is rather the consensus algorithm, which is the major differentiator between the cryptocurrency projects. While Ethereum utilizes a Proof-of-Work, Neo the voting-based delegated Byzantine Fault Tolerance (dBFT), and EOS and TRON use a Delegated Proof-of-Stake (DPoS) algorithm.

Many of the platforms strive to solve scalability problems of popular platforms (Ethereum achieves only 15 transaction per second as maximum) by switching to these newer consensus methods. This also includes the Ethereum developers themselves, who published a wide range of proposals, beginning from a switch of the consensus mechanism to Proof-of-Stake, to sharding (where each node only holds a subset of the state). But later

released platforms often have it easier, as they do not have to care about backwards compatibility, and therefore already test some of those novel ideas.

DApps in Ethereum are quite diverse, and therefore a consensus or security concept for one such DApp might not be the right choice for another. To resolve this, we discussed a proposal with the goal of splitting off every DApp from Ethereum's main chain, and letting them live in their own chain – so called *sidechains*.

The resulting data structure then no longer resembles a chain of transactions, but rather a graph – precisely, a directed acyclic graph (DAG). Several cryptocurrencies already exist that use a DAG for their ledger structure (like *IOTA* and *NANO*), but they do not support smart contracts yet. What makes their support hard is that elements lack the *Totality* axiom in such a construct, making it generally impossible to decide which one of two transactions happened first.

We took a shot at such an implementation, and came up with the *tangleEVM* project, a proof-of-concept smart contract platform based on the *EVM* engine (as that is still the de-facto standard for smart contracts), that builds a separate sidechain for each *DApp* and uses the DAG ledger of *IOTA* – the *Tangle* – as storage layer. The difficulties here were the necessary conversions between those platforms, as they are quite different from each other - e.g. Ethereum smart contracts and contract calls are in bytecode, making it necessary to encode them as tryte strings, as IOTA is based on ternary computing (and is our only option for an already existing DAG ledger). Because IOTA also does not support the reuse of addresses, we looked at different methods of how a chain (or stream) of continuous transactions can be built. For our implementation we chose *Masked Authenticated Messaging*, a module officially supported by IOTA.

We then looked deeper into the Py-EVM, our chosen implementation of the EVM, and the defacto standard Python implementation of the Ethereum protocol. We extended this virtual machine with another *opcode*. Our example simply implemented the *curl* hashing method (used in IOTA), but the gained knowledge can later be used to implement more advanced constructs (e.g. for interoperability with other sidechains). However the most difficult part here was to modify existing smart contract bytecode, so that the newly implemented opcode could be tested. We also needed to extend the key-value LevelDB account and chain database with custom attributes, and tried to parse the database as well – not so easy, when the keys simply are 256-bit *Keccak* hashes, and the values non-human-readable *RLP*-encoded strings.

For our prototype we only used a simple proof-of-authority consensus, which unfortunately makes it completely centralized. We have briefly discussed other options, like employing a *Delegated proof-of-stake* consensus algorithms, but a deeper look into this possibility was out of scope for this thesis, and could be part of future research.

APPENDIX A

Smart contract platforms

#	Name	SC Engine	SC Language	Remarks	Selected
1	Bitcoin	Script	Ivy-lang, Balzac	live, engine not turing-complete (whitelisting)[1]	N
2	Ethereum	EVM	Solidity	live (ch. 3)	Y
3	XRP	Codius	/	not live, halted in 2015[2]	N
4	Litecoin	Script	see Bitcoin	see Bitcoin	N
5	Bitcoin Cash	Script / Spedn[3]	Ivy-lang, Balzac / Spedn	see Bitcoin / not live	N
6	EOS	EOS VM	C, C++	WebAssembly based[4], not decentralized[5]	Y
7	Binance Coin			is a token	N
8	Tether			is a token	N
9	Bitcoin SV	see Bitcoin		see Bitcoin	N
10	TRON	TVM	Solidity	EVM-compatible	Y
11	Cardano	IELE VM	Plutus, Solidity		Y

[1] https://en.bitcoin.it/wiki/Script

[2] https://bitcoinmagazine.com/articles/ripple-discontinues-smart-contract-platform-codius-citing-small-market-1435182153

[3] https://news.bitcoin.com/meet-spedn-a-smart-contract-programming-language-for-bitcoin-cash/

[4] https://github.com/EOSIO/eos-vm

[5] https://cointelegraph.com/news/eos-proves-yet-again-that-decentralization-is-not-its-priority

12	Stellar			limited SC support via SSC, not turing-complete	N
13	UNUS SED LEO			is a token	N
14	Monero			no smart-contracts	N
15	Dash			no smart-contracts	N
16	Chainlink			is a token	N
17	NEO	NeoVM	many (incl. .NET, Java, C++, Python)	live	Y
18	IOTA	Qubic	Abra	DAG, not turing-complete, not live	Y
19	Cosmos		Ethermint (Solidity) etc.	Not live	N
20	Ethereum Classic	EVM	Solidity	Live	Y
21	NEM			smart-contracts are off-chain[53]	N
22	Zcash			no smart-contracts	N
23	Ontology	Native, NeoVM	see NeoVM		N
24	Maker			is a token	N
25	Tezos	Tezos VM	Michelson	functional language, supports formal verification[6]	N
26	Qtum	EVM	Solidity	Live	Y
27	Bitcoin Gold	Script	see Bitcoin	see Bitcoin	N
28	VeChain	EVM + built-ins extension	Solidity	Live	Y
29	Crypto.com Chain			no smart-contracts	N
30	Basic Attention Token			is a token	N
31	Dogecoin			no smart-contracts	N
32	USD Coin			is a token	N
33	OmiseGO			is a token	N
34	BitTorrent			is a token	N
35	Decred		Decred transaction scripts	not turing-complete, similar to Bitcoin[7]	N

[6] https://www.apriorit.com/dev-blog/602-tezos-blockchain-smart-contract-overview

[7] https://godoc.org/github.com/decred/dcrd/txscript

36	V Systems			not turing-complete (in the beginning, not live)[8]	
37	Holo			is a token	N
38	TrueUSD			is a token	N
39	Ravencoin			no smart-contracts[9]	N
40	Bitcoin Diamond	Script	see Bitcoin	see Bitcoin	N
41	Lisk	n/a	JavaScript	see 2.8	Y
42	Pundi X			is a token	N
43	Egretia			is a token	N
44	Huobi Token			is a token	N
45	HedgeTrade			is a token	N
46	Aurora			is a token	N
47	HyperCash	Hcd	/	not live, research compatibility with EVM and EOS[10]	N
48	Waves	Waves	RIDE	not live	N
59	0x			is a token	N
50	NANO			DAG, no smart-contracts	N

Table A.1: Top50 Cryptocurrencies platforms, data from coinmarketcap.com, snapshot from 30 June 2019[12]

[8] https://medium.com/vsystems/v-systems-smart-contract-and-token-system-206bef9d67b2

[9] https://www.reddit.com/r/Ravencoin/comments/9npsic/doeswill_raven_have_smart_contracts_like_eth_what/

[10] https://medium.com/@media_30378/weekly-development-update-82adfe0e8ab7

APPENDIX B

Ethereum snippets

```
import rlp
from eth_keys.datatypes import PrivateKey
from eth_typing import Address
from eth_utils import decode_hex
from eth.vm.forks.istanbul.transactions import IstanbulUnsignedTransaction

private_key = PrivateKey(decode_hex(
    '0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8'
))
public_key = private_key.public_key
canoncial_address = public_key.to_canonical_address()
address = public_key.to_address()
checksum_address = public_key.to_checksum_address()

unsigned_tx = IstanbulUnsignedTransaction(
  nonce=0,
  gas_price=0,
  gas=5000000,
  to=Address(b''),
  value=0,
  data=decode_hex('18160ddd'),
)

signed_tx = unsigned_tx.as_signed_transaction(private_key, chain_id=1908)

print("private key: %s" % private_key)
print("public key: %s" % public_key)
print("canoncial address: %s" % canoncial_address)
print("address: %s" % address)
print("checksum address: %s" % checksum_address)
print("unsigned tx: %s" % rlp.encode(unsigned_tx))
print("signed tx: %s" % rlp.encode(signed_tx))
```

Listing 5: Deriving an Ethereum address and RLP transaction

```
# vm/mnemonics.py
CURL = 'CURL'
```

```
# vm/opcode_values.py
CURL = 0xe0

# vm/opcodes.py
import copy
from eth.vm.logic import sha3
from eth.vm.opcode import as_opcode
from eth_utils.toolz import merge
from eth import constants
from eth.vm.forks.petersburg.opcodes import PETERSBURG_OPCODES
from tanglevm.vm import opcode_values, mnemonics
from tanglevm.vm.logic import curl

UPDATED_OPCODES = {
  opcode_values.CURL: as_opcode(
    logic_fn=curl.curl,
    mnemonic=mnemonics.CURL,
    gas_cost=constants.GAS_SHA3,
  ),
}

TANGLEVM_OPCODES = merge(
  copy.deepcopy(PETERSBURG_OPCODES),
  UPDATED_OPCODES,
)

# vm/logic/curl.py
from eth_hash.auto import keccak
from eth_utils import encode_hex, decode_hex
from iota import TryteString
from iota.crypto.kerl import conv, Kerl
from iota.crypto import Curl

from eth import constants
from eth._utils.numeric import ceil32
from eth.vm.computation import BaseComputation
from tanglevm.helper import encodeBytesAsTryteString,
   decodeTryteStringFromBytes, encodeTryteStringAsBytes

def curl(computation: BaseComputation) -> None:
  start_position, size = computation.stack_pop_ints(2)
  computation.extend_memory(start_position, size)
```

```
    curl_bytes = computation.memory_read_bytes(start_position, size)
    word_count = ceil32(len(curl_bytes)) // 32

    gas_cost = constants.GAS_SHA3WORD * word_count
    computation.consume_gas(gas_cost, reason="CURL: word gas cost")

    trytes = decodeTryteStringFromBytes(curl_bytes)
    trits = conv.trytes_to_trits(trytes)
    kerl = Kerl()
    kerl.absorb(trits)
    trits_out = []
    kerl.squeeze(trits_out)
    trytes_out = conv.trits_to_trytes(trits_out)
    bytes_result = encodeTryteStringAsBytes(trytes_out)
    computation.stack_push_bytes(bytes_result[18:])
    computation.stack_push_bytes(b'\x00' * 14 + bytes_result[:18])

# vm/computation.py
from eth.vm.forks.petersburg.computation import PETERSBURG_PRECOMPILES,
  PetersburgComputation
from .opcodes import TANGLEVM_OPCODES

TANGLEVM_PRECOMPILES = PETERSBURG_PRECOMPILES

class TanglevmComputation(PetersburgComputation):
    # Override
    opcodes = TANGLEVM_OPCODES
    _precompiles = TANGLEVM_PRECOMPILES

# vm/state.py
from eth.vm.forks.petersburg import PetersburgState
from tanglevm.vm.computation import TanglevmComputation

class TanglevmState(PetersburgState):
    computation_class = TanglevmComputation
```

Listing 6: Implementation of the *CURL* opcode in the tangleVM

APPENDIX C

Solidity snippets

```
pragma solidity ^0.5.0;

import "./openzeppelin-solidity/contracts/token/ERC20/ERC20.sol";
import "./openzeppelin-solidity/contracts/token/ERC20/ERC20Detailed.sol";

contract SuperToken is ERC20, ERC20Detailed {

  constructor () ERC20Detailed("SuperToken", "SUP", 18) public {
    _mint(msg.sender, 4919 * (10 ** uint256(decimals())));
  }
}
```

Listing 7: Source code of SuperToken

```
pragma solidity ^0.5.0;

interface IERC20 {
  function totalSupply() external view returns (uint256);
  function balanceOf(address account) external view returns (uint256);
  function transfer(address recipient, uint256 amount)
            external returns (bool);
  function allowance(address owner, address spender)
            external view returns (uint256);
  function approve(address spender, uint256 amount) external returns (bool);
  function transferFrom(address sender, address recipient, uint256 amount)
            external returns (bool);

  event Transfer(address indexed from, address indexed to, uint256 value);
  event Approval(address indexed owner, address indexed spender,
                uint256 value);
}
```

Listing 8: Interface ERC-20 compatible smart contracts need to implement[1]

[1] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol

List of Figures

List of Tables

Acronyms

DAG Directed acyclic graph. 5–11, 19, 42, 44, 46–48, 52, 56, 66, 67, 76, 80, 91

DPoS Delegated Proof-of-Stake. 16

ECDSA Elliptic Curve Digital Signature Algorithm. 26

EVM Ethereum Virtual Machine. 8, 14–17, 22, 27–29, 32, 44, 48, 49, 51, 54–56, 63, 64, 66–69, 71, 76, 79, 80

eWASM Ethereum WebAssembly. 16, 26

IRI IOTA Reference Implementation. 46, 52, 64

MAM Masked Authentication Messaging. 56–61, 76, 77, 92

PoC Proof of Concept. 56, 64

PoS Proof-of-Stake. 1, 15, 26, 79

PoW Proof-of-Work. 1, 26, 65, 79

W-OTS Winternitz one-time signature scheme. 42, 60

Bibliography

[1] URL: https://docs.iota.org/docs/the-tangle/0.1/concepts/the-coordinator.

[2] ABmushi. *IOTA: MAM Eloquently Explained*. Feb. 2018. URL: https://medium.com/coinmonks/iota-mam-eloquently-explained-d7505863b413.

[3] Jakob Ackermann and Maximilian Meier. "Blockchain 3.0 - The next generation of blockchain systems". In: (Sept. 2018).

[4] Adam Back et al. "Enabling Blockchain Innovations with Pegged Sidechains". In: 2014.

[5] Marco Bareis. "Comparison of Ethereum and NEO as smart contract platforms". In: (Oct. 2019).

[6] Massimo Bartoletti and Livio Pompianu. "An empirical analysis of smart contracts: platforms, applications, and design patterns". In: *10323 LNCS, Financial Cryptography and Data Security*. Springer. 2017, pp. 494–509. URL: http://arxiv.org/abs/1703.06322.

[7] Steven Burg. *Consensus algorithms for socially responsible blockchains*. July 2019. URL: http://www.flabizlaw.org/files/Consensus%20Algorithms%20Steve%20Burg.pdf.

[8] Vitalik Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. https://github.com/ethereum/wiki/wiki/White-Paper. 2013.

[9] Vitalik Buterin. *Skinny CREATE2*. URL: http://eips.ethereum.org/EIPS/eip-1014.

[10] Jakub Cech. *IRI 1.6.0 with local snapshots out now!* Jan. 2019. URL: https://blog.iota.org/iri-1-6-0-with-local-snapshots-out-now-fc4d991faba8.

[11] ChainSecurity. *Constantinople enables new Reentrancy Attack*. Jan. 2019. URL: https://medium.com/chainsecurity/constantinople-enables-new-reentrancy-attack-ace4088297d9.

[12] *coinmarketcap.com Historical Snapshot - 30 June 2019*. June 2019. URL: https://coinmarketcap.com/historical/20190701/.

[13] Lin William Cong and Zhiguo He. *Blockchain disruption and smart contracts*. Tech. rep. National Bureau of Economic Research, 2018.

[14] *Contract ABI Specification*. URL: https://solidity.readthedocs.io/en/develop/abi-spec.html.

[15] Tim Copeland. *Decrypt Guide: The future of Ethereum*. Apr. 2019. URL: https://decryptmedia.com/6219/decrypt-guide-future-of-ethereum.

[16] Chris Coverdale. *Solidity: Transaction-Ordering Attacks Coinmonks Medium*. Mar. 2018. URL: https://medium.com/coinmonks/solidity-transaction-ordering-attacks-1193a014884e.

[17] cr0ssing. *cr0ssing/raam.client.js*. July 2019. URL: https://github.com/cr0ssing/raam.client.js.

[18] Kyle Croman et al. "On Scaling Decentralized Blockchains". In: vol. 9604. Feb. 2016, pp. 106–125. ISBN: 978-3-662-53356-7. DOI: 10.1007/978-3-662-53357-4_8.

[19] Monika Di Angelo and Gernot Salzer. "Mayflies, Breeders, and Busy Bees in Ethereum: Smart Contracts Over Time". In: *Third ACM Workshop on Blockchains, Cryptocurrencies and Contracts (BCC'19)*. ACM Press. 2019.

[20] *Ethereum Virtual Machine Opcodes*. URL: https://ethervm.io/.

[21] *Ethereum vs EOS vs Cardano vs Rootstock - Smart Contract Platform Comparison and Review " The Merkle Hash*. Oct. 2019. URL: https://themerkle.com/eth-eos-ada-rsk-comparison/.

[22] Alon Gal. *The Tangle: an illustrated introduction - Part 2: transaction rates, latency, and random walks*. Feb. 2018. URL: https://blog.iota.org/the-tangle-an-illustrated-introduction-c0a86f994445.

[23] Alon Gal. *The Tangle: an illustrated introduction - Part 3: Cumulative weights and weighted random walks*. Feb. 2018. URL: https://blog.iota.org/the-tangle-an-illustrated-introduction-f359b8b2ec80.

[24] Alon Gal. *The Tangle: an illustrated introduction - Part 4: Approvers, balances, and double-spends*. Feb. 2018. URL: https://blog.iota.org/the-tangle-an-illustrated-introduction-1618d3e140ad.

[25] Alon Gal. *The Tangle: an illustrated introduction - Part 5: Consensus, confirmation confidence, and the coordinator*. Feb. 2018. URL: https://blog.iota.org/the-tangle-an-illustrated-introduction-79f537b0a455.

[26] *Gartner Identifies the Top 10 Strategic Technology Trends for 2019*. URL: https://www.gartner.com/en/newsroom/press-releases/2018-10-15-gartner-identifies-the-top-10-strategic-technology-trends-for-2019.

[27] Peter Gai, Aggelos Kiayias, and Dionysis Zindros. "Proof-of-stake sidechains". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 139–156.

[28] Andrew Greve. *IOTA Announces Coordicide Solution*. June 2019. URL: https://blog.iota.org/coordicide-e039fd43a871.

[29] Bertoni Guido et al. *Cryptographic sponge functions*. 2011.

[30] Li Haifeng. *How We're Designing a Better Virtual Machine than Ethereum and EOS*. Sept. 2018. URL: https://hackernoon.com/how-were-designing-a-better-virtual-machine-than-ethereum-and-eos-7b60ba62fc5a.

[31] Paul Handy. *Introducing Masked Authenticated Messaging*. Nov. 2017. URL: https://blog.iota.org/introducing-masked-authenticated-messaging-e55c1822d50e.

[32] Hashgard. *Deep Analysis of VM: What virtual machines are used by Ethereum and EOS?* July 2019. URL: https://medium.com/@hashgard/deep-analysis-of-vm-what-virtual-machines-are-used-by-ethereum-and-eos-af925b9408a3.

[33] Ethan Heilman et al. "Cryptanalysis of Curl-P and Other Attacks on the IOTA Cryptocurrency." In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 344.

[34] Lukas Hetzenecker. *Integrating my Smart Home into the Tangle*. Oct. 2018. URL: https://medium.com/@lukashetzenecker/integrating-my-smart-home-into-the-tangle-d88ae03eb9bb.

[35] Eric Hop. *Exploring the IOTA signing process*. Apr. 2018. URL: https://medium.com/iota-demystified/exploring-the-iota-signing-process-eb142c839d7f.

[36] Matt Hussey. *CryptoKitties*. Mar. 2019. URL: https://decryptmedia.com/resources/cryptokitties.

[37] IOTA foundation. *The Qubic Protocol*. URL: https://qubic.iota.org/protocol.

[38] Iota-Community. *iota-community/one-command-tangle*. July 2019. URL: https://github.com/iota-community/one-command-tangle.

[39] *IRI Documentation: IRI configuration options*. URL: https://docs.iota.org/docs/node-software/0.1/iri/references/iri-configuration-options.

[40] *IRI Documentation: ZMQ events*. URL: https://docs.iota.org/docs/node-software/0.1/iri/references/zmq-events.

[41] Nick Johnson. *Why I find Iota deeply alarming*. Sept. 2017. URL: https://hackernoon.com/why-i-find-iota-deeply-alarming-934f1908194b.

[42] Jon Jordan, Ian Kane, and Modesta Jurgeleviien. *DappRadar 2019 dapp Industry Review*. Dec. 2019. URL: https://dappradar.com/blog/dappradar-2019-dapp-industry-review.

[43] Grenoble Ecole de Management. Kariappa Bheemaiah. *Block Chain 2.0: The Renaissance of Money*. Aug. 2015. URL: https://www.wired.com/insights/2015/01/block-chain-2-0/.

[44] Aggelos Kiayias and Dionysis Zindros. "Proof-of-Work Sidechains". In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 1048.

[45] Robin Lamberti. *Random Access Authenticated Messaging*. Feb. 2019. URL: https://blog.usejournal.com/random-access-authenticated-messaging-45a5f40f2532.

[46] Lekkertech. *IOTA Signatures, Private Keys and Address Reuse?* URL: http://blog.lekkertech.net/blog/2018/03/07/iota-signatures/.

[47] Sergio Demian Lerner. *RSK White Paper*. Jan. 2019. URL: https://www.rsk.co/Whitepapers/RSK-White-Paper-Updated.pdf.

[48] John D. Licciardello. *The First Cohort of Ecosystem Development Fund Grantees*. Aug. 2018. URL: https://blog.iota.org/the-first-cohort-of-ecosystem-development-fund-grantees-e9da89ecfb56.

[49] Robert Lie. *IOTA Quickguide Tutorial*. URL: https://www.mobilefish.com/developer/iota/iota_quickguide_tutorial.html.

[50] Chunming Liu, Daniel Wang, and Ming Wu. *Vite Whitepaper*. URL: https://github.com/vitelabs/whitepaper.

[51] LiveOverflow. *Ethereum Smart Contract Code Review #1 - Real World CTF 2018*. Dec. 2018. URL: https://www.youtube.com/watch?v=ozqOlUVKL1s.

[52] Mitchell Moos. *IOTA Halts for 15 Hours from Coordinator Bug*. Dec. 2019. URL: https://cryptobriefing.com/iota-halts-15-hours-from-coordinator-bug/.

[53] Michiel Mulders. *Comparison of Smart Contract Platforms*. Mar. 2018. URL: https://hackernoon.com/comparison-of-smart-contract-platforms-2796e34673b7.

[54] Nanocurrency. *nanocurrency/raiblocks*. URL: https://github.com/nanocurrency/raiblocks/wiki/Block-lattice.

[55] Arvind Narayanan et al. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton, NJ, USA: Princeton University Press, 2016. ISBN: 0691171696, 9780691171692.

[56] Neha Narula. *Cryptographic vulnerabilities in IOTA*. Sept. 2017. URL: https://medium.com/@neha/cryptographic-vulnerabilities-in-iota-9a6a9ddc4367.

[57] Neo. *Upgrade of NeoVM NEO Smart Economy Medium*. Dec. 2018. URL: https://medium.com/neo-smart-economy/upgrade-of-neovm-36ee232835d9.

[58] *NEO White Paper*. URL: https://docs.neo.org/en-us/whitepaper.html.

[59] Open Trading Network. *How Crypto-Kitties Disrupted the Ethereum Network Hacker Noon*. Dec. 2017. URL: https://hackernoon.com/how-crypto-kitties-disrupted-the-ethereum-network-845c22aa1e6e.

[60] OpenZeppelin. *Ethereum in Depth, Part 2*. Dec. 2019. URL: https://blog.openzeppelin.com/ethereum-in-depth-part-2-6339cf6bddb9/.

[61] Overtorment. *Overtorment/awesome-smart-contracts*. URL: https://github.com/Overtorment/awesome-smart-contracts.

[62] Jan Pauseback. *EDAG and d-fine develop a Smart Parking Ecosystem, powered by IOTA*. Sept. 2019. URL: https://blog.iota.org/edag-and-d-fine-develop-a-smart-parking-ecosystem-powered-by-iota-afdae5641089.

[63] Carlos Pérez Jiménez. "Analysis of the Ethereum state". PhD thesis. Universitat Oberta de Catalunya, 2018. URL: http://openaccess.uoc.edu/webapps/o2/bitstream/10609/74445/6/cperezjimenezTFM0118memoria.pdf.

[64] Serguei Popov. *On the timestamps in the tangle*. https://assets.ctfassets.net/r1dr6vzfxhev/4XgiKaTkUgEyW8O8qGg6wm/32f3a7c28022e35e4d5d0e858c0973a9/On_the_timestamps_in_the_tangle_-_20182502.pdf. 2018.

[65] Serguei Popov. *The Tangle*. https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218e1ec/iota1_4_3.pdf. 2018.

[66] *r/Iota - Have Smart Contracts been abandoned with Qubic?* URL: https://www.reddit.com/r/Iota/comments/by8p4w/have_smart_contracts_been_abandoned_with_qubic/.

[67] Alejandro Reyes. *Ethereum - The World Computer*. Feb. 2018. URL: https://medium.com/@reyesale/ethereum-the-world-computer-fb7b58948280.

[68] Vaibhav Saini. URL: https://hackernoon.com/difference-between-sidechains-and-state-channels-2f5dfbd10707.

[69] William Sanders. *Posets and Consensus*. Jan. 2019. URL: https://blog.iota.org/posets-and-consensus-fe4c034595ab.

[70] Ale Santander. *Deconstructing a Solidity Contract -Part I: Introduction*. July 2019. URL: https://blog.openzeppelin.com/deconstructing-a-solidity-contract-part-i-introduction-832efd2d7737/.

[71] Ondrej Sarnecký. *Solving scalability of Ethereum through Loom Sidechains (Tutorial)*. URL: https://hackernoon.com/solving-scalability-of-ethereum-through-loom-sidechains-tutorial-2837307d454.

[72] Dominik Schiener. *Bundles*. URL: https://domschiener.gitbooks.io/iota-guide/content/chapter1/bundles.html.

[73] *Simple Facts About EOS*. Oct. 2018. URL: https://medium.com/@dappdotcom/simple-facts-about-eos-a090e9a591ed.

[74] Alexey Sobolev. *Integrate Hyperledger Fabric with the IOTA Tangle*. Nov. 2019. URL: https://blog.iota.org/integrate-hyperledger-fabric-with-the-iota-tangle-9bc3ac873e82.

[75] David Sønstebø. *Curl disclosure, beyond the headline*. Apr. 2018. URL: https://blog.iota.org/curl-disclosure-beyond-the-headline-1814048d08ef.

[76] *Structure of a bundle - IOTA Documentation*. URL: https://docs.iota.org/docs/dev-essentials/0.1/references/structure-of-a-bundle.

[77] syedjafri. *Detailed list of changes made to EOS Technical Whitepaper*. Mar. 2018. URL: https://busy.org/@syedjafri/detailed-list-of-changes-made-to-eos-technical-whitepaper.

[78] Nick Szabo. 1996. URL: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[79] Nick Szabo. "Smart Contracts : Building Blocks for Digital Markets". In: 2018.

[80] Hans Rudolf Trüeb. "Smart Contracts". In: (2018), pp. 701–712. URL: https://archive.is/X3lR2.

[81] Saini Vaibhav. *Getting Deep Into EVM: How Ethereum Works Backstage*. URL: https://hackernoon.com/getting-deep-into-evm-how-ethereum-works-backstage-ac7efa1f0015.

[82] Vbakke. *vbakke/trytes*. Jan. 2018. URL: https://github.com/vbakke/trytes.

[83] Wolfgang Welz. *Assuring authenticity in the Tangle with signatures*. Feb. 2019. URL: https://blog.iota.org/assuring-authenticity-in-the-tangle-with-signatures-791897d7b998.

[84] Ethereum wiki. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. Tech. rep. 2017, pp. 1–23. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[85] Joon Ian Wong. *CryptoKitties is jamming up the ethereum network*. Dec. 2017. URL: https://qz.com/1145833/cryptokitties-is-causing-ethereum-network-congestion/.

[86] Jesse Yli-Huumo et al. "Where Is Current Research on Blockchain Technology?A Systematic Review". In: *PloS one* 11.10 (Oct. 2016), pp. 1–27. DOI: 10.1371/journal.pone.0163477. URL: https://doi.org/10.1371/journal.pone.0163477.