TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

Bachelor's Thesis in Informatics

Analysis of the Solidity Compiler for Smart-Contract Redundancy Detection

Jonas Gebele

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

Bachelor's Thesis in Informatics

Analysis of the Solidity Compiler for Smart-Contract Redundancy Detection

Analyse des Solidity Compilers zur Smart-Contract Redundanzerkennung

Author:	Jonas Gebele
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Ulrich Gallersdörfer, M.Sc.
Submission Date:	November 16, 2020

Acknowledgments

First and foremost, I would like to thank my thesis advisor, Ulrich Gallersdörfer. Without his assistance and dedicated support within every step throughout the process of creating this thesis, this paper would have never been accomplished. I am especially grateful for the time he took for our initial discussions of the research proposal.
I would also like to show gratitude to Prof. Dr Florian Matthes for the opportunity to write this thesis at his supervision at the chair for Software Engineering for Business Information Systems. During the discussion of the definition of the topic and the kick-off meeting he raised many precious points and I hope that I have managed to address several of them here.
Likewise, I want to thank Dr. Christian Reitwießner, who was willing to support this thesis by providing technical details and answering questions regarding the internals of the Solidity Compiler in a conversation with us.
Most importantly, none of this could have happened without my family. I thank them for their encouragement throughout my studies, especially during the last weeks of my thesis. I also appreciate that my family member Olya Gebele has always taken the time to proofread my work.

Abstract

Smart-Contracts based on decentralized blockchain-systems like Ethereum are one of the most promising emerging technologies that become more and more attractive while enabling completely new types of applications. Solidity, the most popular programming-language for smart-contract development, is regularly maintained and improved by a large community and the Solidity-Commandline-Compiler is integrated in a variety of development-environments for decentralized applications. However, how the compiler and the optimization-processes of the bytecodes work internally is unfortunately not easy to comprehend for the majority of developers. This bachelor thesis will shed light onto this topic by taking a closer look at the concrete implementation of the Solidity compiler and the involved optimizers.
In this thesis we will analyze the source-code of the Solidity compiler and look at the different optimization-procedures and how they affect the bytecode. By first determining the overall structure of EVM-bytecode and describing the functionality of the individual bytecode-segments, we then go into greater detail about the modifications in the different sections due to bytecode-optimization.
Furthermore, we explain how the different optimization-features of the compiler-instructions internally impact the optimizer-utilization in the compiler and how the different optimization influences the resulting bytecode and which modifications can be identified.
We then look at the implications of different bytecode-optimization and hereby analyze how to detect potentially redundant bytecodes on the Ethereum blockchain due to lack of optimization. In this context we introduce a design-proposal of such an attempt and discuss conceptional and technical challenges.

Contents

1 Introduction

1.1 Motivation

Smart-Contracts have become a household-name for an emerging technology. Over the last few years, the space has evolved from a small experimental niche-technology to a multi-billion dollar ecosystem on the Ethereum network. A large variety of applications, until now predominantly of financial nature, handle assets in the 13 billion US dollar range[1] as of the year 2020. In doing so, attempts are being made to replicate excising services from the traditional financial-industry with the use of smart-contracts on a blockchain-based system like Ethereum and thus reshape them in a simpler, more transparent and fairer way. Examples for such services are decentralized exchanges, derivative platforms or lending-protocols. The trust-less execution of smart-contracts through a network of computers synchronized by consensus mechanisms aims to replace the need of middle-man for financial transaction in a digital age. Thus, the trust and functionality depends solely on the correct execution of the smart-contract bytecodes that are stored on the Ethereum blockchain.
Until this bytecode is transformed from a high-level programming language into an executable bytecode, a compiler has to perform several seemingly obscure operations. Although the compiler is maintained and refined regularly by a large community of developers and has been integrated into a variety of development-environments and has compiled the majority of smart-contracts that administer billions of dollars, there is no comprehensive documentation on what exactly happens at each step in the compiler.
One special component of the compiler, which will be one of the main topics of the thesis, is the optimization of the smart-contract bytecodes with regard to their execution-costs. The most popular smart-contract alone payed over 12 million dollars last month in execution-costs to the Ethereum network[2].
The use of this inconspicuous optimizer when generating smart-contract bytecodes with the compiler has many mostly unknown effects on other research-domains. These include for example bytecode-analytics, which is becoming increasingly relevant as the size of the Ethereum ecosystem grows rapidly. Most studies, however, do not take the impact of the Solidity optimizer into consideration when determining for example the set of unique smart contracts. However, these assumptions of unique smart-contracts are used in the field of security-analytics for security audits and vulnerability research, in quantitative-analytics for transaction tracking or in usage-analytics for analyzing smart-contract interactions.
In this bachelor thesis we will therefore take a closer look at the compiler and the optimizer

[1]https://defipulse.com/
[2]https://ethgasstation.info/

and shed light on the still relatively unknown impacts and implications of the Solidity optimization.

1.2 Research Questions

In this thesis we will take a closer look at the different optimizers of the Solidity compiler and their effects on the compiled bytecode. Furthermore, implications of modifications in the bytecode through the optimizers are evaluated. We identify the following three research questions, which we aim to address in this bachelor thesis. In order to provide structure to our thesis, we will try to address these three questions consecutively and, where necessary, include relevant background information.

1. **What are the internal workings of the bytecode-optimizers in the Solidity compiler?**

   A detailed documentation for the internals of the compiler with respect to optimization will be provided, which does not yet exist in such an comprehensive form.

   a) Of which sub-optimizers does the bytecode-optimizer of the Solidity compiler consist and what is their functionality?

   b) Which bytecode-sections get optimized in what way of the compilation-process?

2. **How does enabling the optimization in the compiler-instruction modify the bytecode in general?**

   On which bytecode-sections and opcode-patterns can we identify optimizations by the bytecode-optimizer and distinguish therefore optimized bytecodes from non-optimized bytecodes?

   a) How do the compilation-parameters affect the optimization-process of the compiler and the resulting bytecode?

   b) Which opcode-patterns and bytecode-methods can be simplified through setting optimization in the compiler-instruction?

3. **How many bytecodes and therefore smart-contracts are redundant regarding their functionality due to different or missing optimization?**

   Redundancies in smart-contract bytecode-sets of the Ethereum blockchain are to be detected in order to improve the quality of existing data sets.

   a) What design could a re-optimizer have and what restrictions on the re-optimization are there?

   b) In which way do the bytecode-sets being analyzed need to be limited?

1.3 Methodology

The work presented can be divided into to following phases: In the beginning of the thesis in our discovery-phase, the questions regarding the bytecode-optimizers were analyzed and the resulting redundancies on the Ethereum blockchain were discussed. Following, background and literature research was conducted on the relevant topics regarding the fundamentals of Ethereum, the EVM-bytecode and the Solidity compiler. Afterwards, the Solidity compiler and its functioning was studied in detail directly from debugging an actual build-target from the implementation of the Solidity compiler (version 0.5.3) and resulting modifications of the bytecode-sections were analyzed. Last but not least a feasible design and resulting challenges of such a re-optimizer to detect redundancy of bytecode due to missing optimization, were discussed. Finally, the research findings were consolidated and additional research-objectives were formulated.

1.4 Structure

The rest of this thesis is structured as the following: First, in chapter 2, we provide essential background information on Ethereum, the Ethereum Virtual Machine and EVM-bytecode in order to be able to address our research questions, which will be referred to throughout the entire thesis. In chapter 3, we analyze the different optimizers of the Solidity compiler and describe how to use them with the compiler-instructions of the Solidity-commandline-compiler. Then, in chapter 4, we discuss the individual bytecode-sections and what functionalities they provide for the EVM. Afterwards, we describe the effects of the several optimization-methodologies on the different bytecode-segments and opcode-patterns in chapter 5, where we discuss in detail, how differences in optimization can be identified in the bytecode. In chapter 6 we provide a feasible design for a re-optimizer to detect potential redundancies in the bytecodes on the Ethereum blockchain due to missing bytecode-optimization. Finally, we summarize our results of this thesis and propose follow-up research-questions in chapter 7.

2 Ethereum Foundations

2.1 Ethereum Fundamentals

Ethereum is a blockchain-based system operating on a distributed computing-network. The open-source software-system Ethereum has several components that make the system unique and enable new kinds of innovative applications. One of these components is a peer-to-peer (P2P) network over which all participants share messages and new states. This exchanged information is interpreted internally as messages. At its core, each participant operates a state machine, which is synchronized between all participants in a transparent way following a consensus-mechanism. This consensus-mechanism (currently proof-of-work as of November 2020) is based on a game-theoretical incentive scheme that enforces cooperation among all participants in a network of rational economic actors. Several different clients of the network were developed independently of each other and are currently part of the network [1].
In contrast to other decentralized platforms, Ethereum is a general-purpose blockchain build for decentralized applications. Unlike Bitcoin, as an example, Ethereum can easily change its state through a special designed high-level programming language [2].
The native currency of Ethereum (that makes the game-theoretical incentive-structure work) is called ether (ETH). Ether can be subdivided into 10^{18} Wei, or alternatively 10^{9} Gwei [3].
Central to the Ethereum network are wallets, a application that, among other things, stores a public and private key-pair and thereby authorizes transactions in the Ethereum network for a corresponding account [2].
The next section deals with transactions in more detail. Topics such as Elliptic-Curve Cryptography and the functionality of the consensus-mechanism are central components of the Ethereum network, but play a secondary role to the research questions dealt with in this thesis. For this reason, these and other fundamentals of Ethereum not relevant to this thesis will not be discussed any further.

2.2 Ethereum Transactions

From a technical point of view, a transaction is just a cryptographically signed instruction for the EVM that can change its singleton state. Two types of transactions can be distinguished: on the one hand transactions which are instruction-calls in the form of message-calls for the Ethereum Virtual Machine to change the state; and on the other hand, transactions which are used to deploy a new smart-contract code to an associated account (see section 2.1) [2].

2.2.1 Transaction Components

The components of an Ethereum transaction were specified in the original Ethereum Yellowpaper by Gavin Wood [2]. A transaction usually contains the following elements:

- The **nonce** represents the number of transactions confirmed on the blockchain that were initiated by the account and is calculated dynamically. A nonce, which, in contrast to Bitcoin, is necessary in an account-based protocol like Ethereum to prioritize and manage transactions and to uniquely identify transactions to prevent duplicate transactions.

- The **gasPrice** is the number of Wei-units that the sender is willing to pay per unit of Gas (see subsection 2.4.5). This allows the originator of an instruction to determine how much he wants to pay for the execution of an instruction, for a defined number of gas needed. The higher the price (in Gwei) paid per Gas, the faster the transaction is executed and, therefore, confirmed.

- The **gasLimit** is a specified fixed amount of Gas (see subsection 2.4.5), which the initiator is willing to spend at most in order to execute the instructions. This is especially useful because before a smart-contract is executed, the number of instructions, that are going to be executed, can never be determined in advance.

- The recipient, which is indicated by the label **to**. This can be a 160-bit address of the recipient of the instruction-call or in case of a code-creation transaction the zero address.

- The **value** element defines the number of Wei-units to be transferred between the accounts of the sender and the recipient of the instruction-call. If the instruction-call is a contract-creation transaction, the smart-contract account is initialized with the amount of Wei.

- The values for **v**, **r**, **s** represent the ECDSA digital signatures of the instruction from the sender that initiated the instruction. For the exact functionality of ECDSA signatures reference is made in the following work [4].

The last element is an array of theoretically unlimited length with a binary data-payload. Its utilization depends on what type of transaction is being made. In case of a contract-creation transaction the element is referred to as *init*, which contains the deployment bytecode. For the exact functionality of the contract-creation transaction, please refer to subsection 2.2.3. However, if the transaction is a message-call instruction, it is referred to as *data* element (as shown in Figure 2.1). In the subsection 2.2.2 we will explain in more detail the standards of the data-payload for a message-call.

The public key of the sender of the instruction call can be inferred from the *v,r,s* components of the ECDSA signature. Because of that the address of the sender does not have to be explicitly specified as an element in the transaction and. Similarly, all the other data (such as the hash of the transaction or the block number in which the transaction is contained) can be

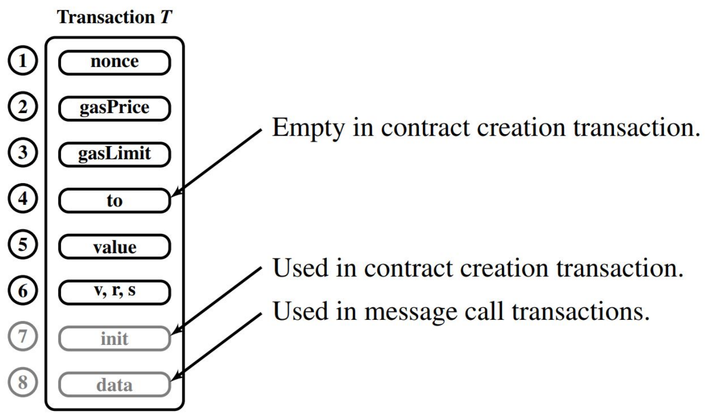

Transaction T
1
nonce
2
gasPrice
3
gasLimit
4
to
5
value
6
v, r, s
7
init
8
data
Empty in contract creation transaction.
Used in contract creation transaction.
Used in message call transactions.

Figure 2.1: An Ethereum transaction consists of 6 default fields and 2 specific payloads that are dependent on the type of transaction [5].

determined dynamically by an Ethereum node and would be therefore redundant in the the transaction-payload [1].

The transaction is serialized in a binary form using the *Recursive Length Prefix* encoding [6]. Due to the specific lengths of the components, the element-names can be dispensed for the raw transaction.

2.2.2 Transaction Data

In transactions to smart-contract accounts, the data-element is interpreted as a *contract-invocation* by the EVM. This contract-invocation is interpreted as a function-invocation by most smart contracts, which corresponds to a function call of the smart contract. The data payload for the function-invocation is used to select a certain function with the function-selector and to pass arguments to the function.
The function-selector consists of the first 4 bytes of its *Keccak-256* hash of the specification of the function name and the data types of the arguments. The arguments are packed in brackets and are separated by commas. This value is hashed as a string and thus can be used as a unique identifier for the function of the smart-contract to invoce, for example:

```
$ web3.sha3("withdraw(uint256)")
'0x2e1a7d4d13322e7b96f9a57413e1525c250fb7a9021cf91d1540d5b69f16a49f'
```

Then, the value (with the web3-client the identifier results to 0x2e1a7d4d) to be passed is appended using the big-endian convention as a hexadecimal-value. This value is padded to the full length of the corresponding data-type that is expected by the smart contract [5].

2.2.3 Contract Creation Transaction

A special type of transaction is the *contract-creation* transaction, which always has to be sent to a specified address, known as the *zero-address* (0x0). The address does not correspond to any EOA or a certain smart-contract account. It is only a signal to the EVM to execute the instruction as a contract-deployment instruction.
This instruction is used to execute the given data-payload from the init element by the EVM, which in this case is the deployment bytecode of a smart-contract (for the detailed structure of the deployment-bytecode, please refer to section 4.2). In essence, when the deployment-bytecode is executed, the runtime-bytecode wrapped in the deployment-bytecode is being stored on the blockchain and the first state of the smart-contract account is being initialized by the constructor in the deployment-bytecode (as mentioned in subsection 2.2.2, the smart-contract with the specified *value* element in the transaction) [1].
In general, transactions are essentially just instructions for the EVM that change the singleton state of the account. For the sake of simplicity, however, in the following we will refer simply to them as transactions.

2.3 Smart Contracts

As indicated in section 2.2, there are two types of accounts in Ethereum. One type are Externally owned accounts (EOAs) that are controlled and managed by a user with the help of a wallet and a corresponding private key. The second type are smart-contract accounts, controlled by software-code and without associated private keys. This software is executed by the EVM and is referred to as a *smart-contract*. These type of accounts perform actions only in response to an incoming instruction as the smart contract gets triggered. Every smart-contract account has its own persistent storage where it stores its own variables and functions (a key-value store with 256-bit keys and 256-bit values). This memory can only be accessed from the smart-contract, but is readable to everyone [7].
Smart contracts have typical characteristics such as immutability, determinism, as well as being typically executable on the EVM. However, the term "contract" does not refer to any legal obligation. Smart contracts are limited in their context to the Ethereum singleton state, the calling transactions and information about the blockchain.
Smart contracts can be deployed on the Ethereum blockchain with a contract creation transaction, as described in subsection 2.2.3. The contract can be identified by an Ethereum address that can be derived from the contract creation transaction. The execution of smart-contract functions is single-threaded and atomic. This means that the instruction either succeeds and takes place or fails and returns to its original state. However, the instruction is still kept in the block chain and the gas used will not be refunded.
Due to the immutability of smart-contracts, the software cannot be modified once deployed. A change of a smart contract can only be achieved by re-deploying it. Deleting a smart contract is possible with the *SELFDESTRUCT* opcode, but the transaction history remains unchanged on the blockchain. In practice, software-patterns are used to update smart-contracts despite their immutability. The restriction is bypassed by using proxy-contracts that reference other

smart contracts, which can be replaced that execute the actual instruction [7].

2.3.1 Ethereum High-Level Languages

Smart contracts are executed by the EVM (see section 2.4). The EVM executes instructions in the form of EVM bytecode (see chapter 4), similar to the Java Virtual Machine (JVM) working with Java bytecode. This has the advantage that the execution of smart contracts remains independent of the native platform and hardware.
This bytecode (both abstractly and in the actual C++ implementation) is a stream of assembly commands that the Ethereum Virtual Machine is capable of executing. It is possible to directly implement the smart-contracts in bytecode in the implementation-process, just like for any other processor or virtual machine. However, the implementation directly in bytecode is much more challenging. Due to the increased complexity, there is also an increased risk of security vulnerabilities, which would be particularly critical for smart-contracts [1].
For the mentioned challenges several compilers have been introduced, which can translate a respective programming language into EVM bytecode. Because of the minimalistic architecture and the technical restrictions of the EVM (see section 2.4), completely new compilers have been built instead of reusing existing programming languages. As a result, a number of special-purpose high-level languages have emerged for programming smart contracts executable on the Ethereum Virtual Machine. There are functional programming languages like LLL as well as imperative programming languages like *Solidity*, *Vyper* or *Serpent*. In the following section the most popular and widely used high-level language will be discussed in more detail [5].

2.3.2 Solidity

Solidity is an imperative high-level quasi turing-complete programming language with a syntax similar to JavaScript. Solidity can be compiled with the Solidity compiler to EVM bytecode. The language is statically typed and supports all established features of common programming languages like object-orientation, inheritance or polymorphism [8].
The Solidity compiler compiles Solidity source-code into EVM bytecode. It also has many additional features like a flexible optimizer (see chapter 3), ABI specification output (see subsection 2.3.3) or meta-data output. As Solidity is constantly being improved, changes are made to the Solidity compiler and development is constantly in progress. This has resulted in different compiler-versions over time. The different Solidity compilers correspond each to a different version of the Solidity programming language, which are not always downward compatible. To deal with this problem, a compiler-directive named *version-pragma* exists in Solidity. This version-pragma defines the compatible compiler-versions for the given Solidity source-code. This *pragma-information* is not compiled by the compiler into EVM bytecode, but is only used to check for compatibility. One instance of the Solidity compiler is the build of the Solidity repository, which is the Solidity-commandline-compiler *solc* [9]. The exact syntax and functionality will not be discussed in this thesis.

2.3.3 Application Binary Interface

An Application Binary Interface (ABI) is a specification of an interface between different software-modules. It specifies which data-structures and which functions can be called from the bytecode, since this information is not directly comprehensible from the encoded bytecode. By contrast, an API defines the interface of a software-module in a high-level programming language such as Solidity. Since smart-contracts are stored as bytecodes on Ethereum blockchain, the ABI is of relevance in this context.
Each smart-contract can be specified with an ABI in order to define which instructions and thus which function-calls are applicable to the smart-contract. This ABI specifies as an array of objects in JSON format which function-calls and which events (events will not be discussed in detail in this thesis) the smart-contract identifiers supports. The ABI describes among other information which arguments are expected by the function and which return-types will be given back [9]. This ABI is typically generated by the Solidity compiler. With the flag *–abi* the Solidity-commandline-compiler can produce it, as shown in the instruction below:

```
$ solc --abi Example.sol
```

2.4 Ethereum Virtual Machine

The *Ethereum Virtual Machine* (EVM) is the central component of the Ethereum protocol, which changes the global singleton state while performing instructions. The EVM is thereby similar to a virtual computation-machine like the Java Virtual Machine (JVM) or the virtual machine of Microsoft's *.NET* framework, which converts bytecode into abstracted virtualized computational operations. In the context of Ethereum, the EVM interprets and executes the deployed bytecode of a smart-contract from a given account. The EVM is not required for simple instructions like a value-transfer instructions [1].
The EVM has a minimalistic stack-based architecture with a word-size of 256 bit. The relatively big word-size allows the EVM to perform hash-operations or elliptic-curve cryptography operations without the need to use more than one stack-element for results of these operations. The virtual stack has a size of 1024 elements on which the main operations are performed. In addition, there are also other memory-sections like the volatile memory and the persistent storage (see subsection 2.4.1 for more details) [2].
The EVM is a virtualization of a machine that is limited to computation and memory-storage. Other virtual machines such as the JVM provide access to the environment like the operating system through system-calls or the actual hardware, which is not possible with EVM. This is an intentional design decision to keep the global singleton state of the EVM as independent as possible from external factors. The EVM also has no hardware-support and no system-interface, so you can interact with the virtual machine completely abstracted from the underlying hardware and operating system. If errors occur on the machine, such as an out-of-gas exception or a stack-underflow, the changes of the atomic instructions to the global singleton state are reversed. With the introduction of the concept of Gas, the EVM is quasi-turing complete (see subsection 2.4.5) [2].

Since there is only a specification of the EVM in the Ethereum Yellowpaper, several implementations in different programming languages have been made and are used in the different clients. Due to the simplicity of the EVM architecture and the formal specification, some of the implementations are quite compact with just over 2000 lines of code [3]. Among others, the most widely used EVM implementations are:

- Py-EVM (Python)
- EVMC (C++, C)
- ethereumjs-vm (JavaScript)
- eEVM (C++)
- Hyperledger Burrow (Go)

The EVM is single-threaded because the instructions are ordered externally by the clients. Therefore, the EVM does not support scheduling capabilities, which is comparable to the execution of JavaScript that is executed fully in sequence [1].

2.4.1 Memory-Sections

The EVM has various memory-sections where it can store data, each with a special purpose. First, as mentioned above, the EVM has a stack with a size of 1024 elements with 256 bit words each. The stack can be accessed with swapping-operations to a maximum of the top 16 elements at the same time. Most of the operations are limited to the top two elements. The stack replaces the need for registers. Operations are executed directly on the stack. Of course it would be also possible to store computational data on one of the other memory-sections. However, this takes much more effort and, thus, costs more Gas (see subsection 2.4.5) [9]. The 256-Bit word-size of the stack-elements are relatively big in order to efficiently support elliptic-curve operations and *Keccak256-Hashes* without the need to use more than one stack-element [2].
The volatile memory of the EVM is used for intermediate storage of values outside the stack and for a more flexible memory management. It is used for each of every message-call instruction and is cleared after every call. The memory can be addressed byte-wise in a linear way, but is limited to addressing 256-bits per call. New 256-bit memory-elements can be dynamically allocated, but the costs increase quadratically for each new memory-element. The volatile memory is initialized with zero-values. This type of memory would be comparable to the RAM of a *Von-Neumann* architecture [10].
To store data to the memory of the EVM the *MSTORE* operation is used which uses two arguments. The first argument is the address of the word in memory where the value will be stored and the second is the value that is to be stored. Both arguments are expected to be on the stack [1] (for more detailed information regarding the Opcode-instructions please refer to subsection 2.5.2).
There is also a permanent memory, which is called storage. A storage is allocated for every

account. This storage is persistent over different function-calls and transactions. As already mentioned in this thesis, the storage is a key-value map that maps 256 bit values to 256 bit values. Access to this storage area is especially complex and therefore particularly expensive in terms of Gas (see subsection 2.4.5). The storage can only be accessed from the associated smart-contract, however the storage is visible to everyone [9].
Compared to the *Von-Neumann* architecture, this memory-section corresponds to the ROM. The bytecode of a smart-contract that gets executed by the EVM is to be stored permanent-memory. In contrary to a virtualized *Von-Neumann* architecture the data is not stored in an accessible and mutable memory-section in the general memory but in a special memory-section, which can be accessed only by certain opcodes and is called the permanent memory-section [2].
Due to this division into stack, volatile memory and permanent memory, this memory-separation looks rather like a type of a virtualized Harvard architecture [2]. The division of memory areas according to Harvard architecture primarily reflects the required security-measurements of the EVM.

2.4.2 Global State

The central task of the EVM is to transfer the global singleton state to another valid global state by means of valid instructions of smart-contract executions. The global state of Ethereum refers to all state accounts of all recorded addresses. The consensus-mechanism of Ethereum (will not be discussed in this thesis) creates a global singleton state in the entire network [5]. Each account has its own information about the balance, the nonce, the permanent storage and the program code (see subsection 2.4.1), which is the bytecode in case of a smart-contract. A EOA is also a normal account and, thus, part of the global state, which has no program code and an empty storage (as seen in Figure 2.2). The account balance indicates the number of Wei that are controlled by the account. The Nonce reflects the number of successful transactions in case of an EOA or the number of smart-contract creation instructions in case of a contract-account [11].

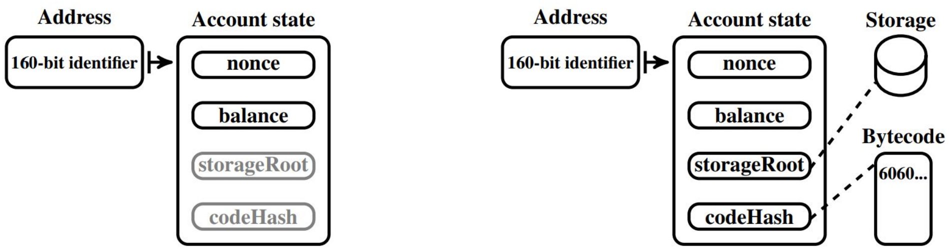


Figure 2.2: State of an EOA on the left and of a contract-account with EVM bytecode storage for contract-data on the right [5].

The permanent storage is the memory section used by smart-contracts, mapping 160-bit words to 160-bit words (see subsection 2.4.1). This information represents in its entirety the

global state of the EVM. Another physical difference in the storage between the accounts is also the hash of storage-trie's root (as shown in Figure 2.2), which can be derived from the transaction and, thus, is not actually a distinctive property of the state of its account [12].
This storage-trie's root is a 256-Bit Keccak-hash value stored in the account storage in the Ethereum state. Also there is the code-hash, a 256-bit hash value of the corresponding bytecode of the account. This code-hash (*codeHash*) is also stored on the Ethereum state. Since the bytecode of an account naturally doesn't change, the code-hash value also doesn't change [5].

2.4.3 State Transition

There are several perspectives on a transition from one state to another state. In the following we shall consider Ethereum from a theoretical point of view, as a transaction-based state machine.
Ethereum can be described as a transaction-based state-machine as described in the Yellow-paper [2]. The global singleton state corresponds to the current state σ of Ethereum. Valid instructions like message-calls or simple value-transfers can change the current state σ to the next global state σ' (as displayed in Figure 2.3).

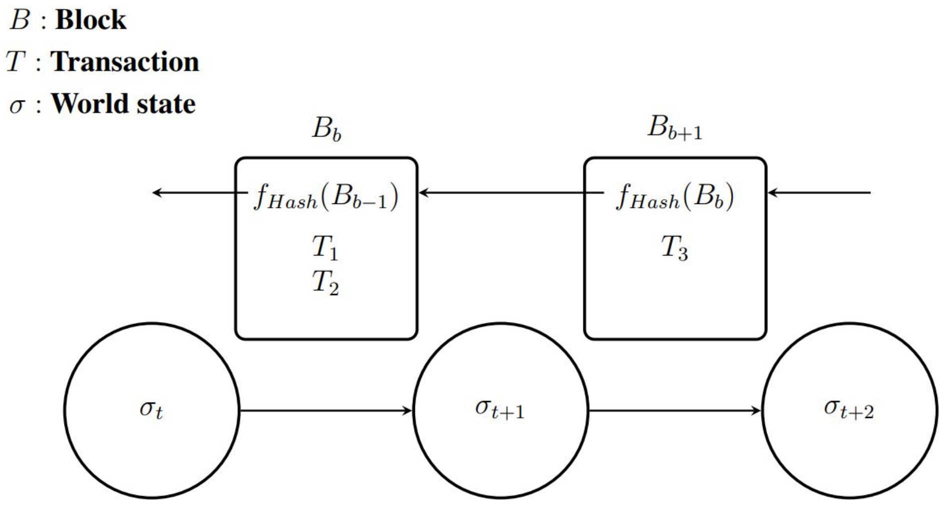


Figure 2.3: Blocks of transitions change the global singleton state through the addition of finalized blocks [5].

This means, transitions between different states are triggered by valid operations of the EVM or value-transfers. From the theoretical point of view, single instructions do not change the global state of Ethereum, as instructions are bundled in groups. According to the actual implementation of Ethereum, only a set of instructions bundled into a block change the global state of Ethereum sequentially, as shown in Figure 2.3. For this reason, Ethereum is rather a sequence of blocks with instructions, where the subsequent block (*child-block*) $\mathrm{B}_b + 1$ refers to the predecessor (*parent-block*) B_b with a hash-reference. The blocks contain the instructions that cause the state to change. More specifically, this means that Ethereum consists of a sequence of blocks that refer to the successor with a hash-value. The blocks contain instructions to change the global singleton state [5].

2.4.4 Execution-Process of the EVM

If a message-call invokes a smart-contract function as an instruction, the EVM is initialized with the account-data as well as the transaction-data and the context of the Ethereum environment (e.g. the current block-information). The bytecode of the smart-contract corresponds to the ROM of the EVM while the permanent storage is the RAM. A program-counter is initialized with zero. The volatile-memory is initialized with zero and the bytecode is loaded into the ROM in the permanent storage.
In general, the EVM processes each instruction of the bytecode sequentially, increases the program-counter after each instruction and then continues with the next instruction of the bytecode until there are no further instructions left. With each instruction the Gas-supply value is subtracted with the respective Gas costs of each instruction of the bytecode. If the gas cost of the transaction exceeds the Gas supply, an out-of-gas exception is thrown and the old state is reinstated. In the case of an error, the nonce of the sender will be increased and the used Gas will be spent, so that a failed instruction will not be completely without effects [1].
Before an instruction is executed and an old state is changed to a new state by a transaction, some basic conditions are being observed. First of all, it is checked if the transaction passes basic conditions, i.e. if the signature is valid, if the nonces of the sender and the recipient are correct and if the value to be transferred is available to spend. Then, the transaction costs are calculated by multiplying the start-Gas with the chosen gasPrice. The address of the sender is derived from the signature. The amount of gas available for the instruction is calculated afterwards. If the account is a contract-account, the bytecode of the smart-contract will be executed. In case the account is an EOA, the account receives the value. If the transaction was successful, the unused gas is returned to the sender of the instruction [13].
The following Figure 2.4 shows an example of a message-call transaction of a smart-contract that changes the global state.

2.4.5 Gas

The EVM is quasi-turing complete with the EVM bytecode that is to be executed. In simplified terms, a programming language is turing-complete, if every possible program can be executed. This would have disastrous consequences for Ethereum and the EVM because then even programs that would never end (e.g. with an endless loop) could be executed on the EVM. Because of the non-resolvability of the halting-problem, it cannot be determined in advance whether a program will terminate, which is why EVM explicitly cannot be turing-complete. Since EVM is a single-threaded virtual machine, a new concept must be introduced to eliminate attack-vectors that attempt to execute non-terminating bytecode.
The solution is to terminate the program without any result after a certain predefined number

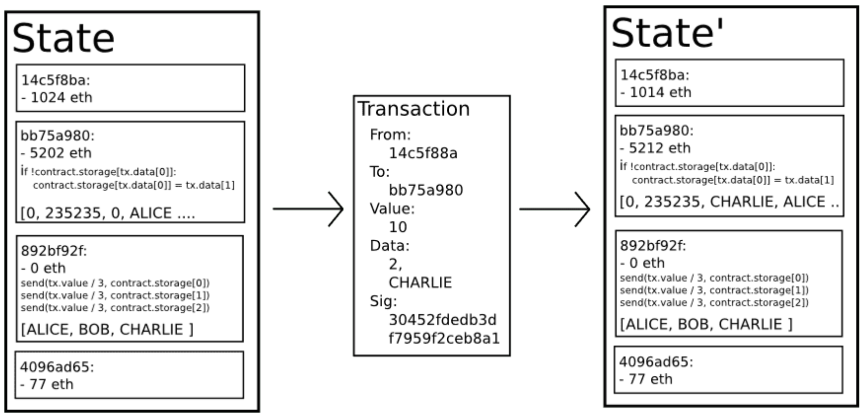

State
14c5f8ba:
- 1024 eth
bb75a980:
- 5202 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, 0, ALICE ....
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth
Transaction
From:
14c5f88a
To:
bb75a980
Value:
10
Data:
2,
CHARLIE
Sig:
30452fdedb3d
f7959f2ceb8a1
State'
14c5f8ba:
- 1014 eth
bb75a980:
- 5212 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, CHARLIE, ALICE ..
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth

Figure 2.4: The Ethereum state transition between two different states takes place while executing a transaction [13].

of computational operations. This limit can be set variably up to a certain maximum (*block-Gas limit*) [1].
Gas is the unit of Ethereum to measure computing power and memory consumption required by the EVM from an operation in bytecode. For each transaction there are two types of gas that are paid. One is a fixed amount of 21,000 gas, which always has to be paid. The other is a variable amount, which depends on how expensive the execution of the bytecode is for the EVM and how much memory is being used. Every single opcode in bytecode (see subsection 2.5.2) is defined by a certain amount of Gas-price. These amounts were decided by the developers of EVM and are separated into different tiers. These range from *zero tier* (0 gas) with the lowest gas costs up to *base tier* (2 Gas), *very low tier* (3 Gas), *low tier* (5 Gas) and *high tier* (10 gas). In addition, there are special tiers, where the costs are subject to more complex rules (for example the instruction *SSTORE*) [14].
It is essential that the Gas values match the actual computing power and memory consumption as close as possible. Changes were made in 2016 as part of the *Ethereum Improvement Proposal* (EIP) 500, where the Gas values of selected input and output-intensive operations had to be adjusted. The mismatched classification of the operations, which were actually much more expensive to perform for the given Gas-value, resulted in *Denial of Service* attacks and, thus, congesting the network for too unfairly insufficient Gas costs [1].
The cost of the entire execution of the instruction is the sum of the cost of each instruction in addition to the fixed 21000 Gas per transaction [14].
Even though the Gas is valued by the computing-power and the memory-consumption of the EVM, there is also a variable price for Gas measured in Ether. With each instruction, the transaction input has to specify how much the sender is willing to pay for the Gas, measured in Gwei. Miners prioritize transactions with a high gas-price, as this is where they earn the most in mining-rewards [1] (more details behind the *consensus mechanism* are not discussed in

detail).
In a transaction, the originator of the instruction specifies a Gas-price measured in Gwei that he is willing to pay and sets a Gas-limit, which is the maximum number of gas units-that the EVM is able to consume before throwing an out-of-Gas exception (see subsection 2.4.4). However, Gas can only be purchased in the context of a transaction, since its unit only exists in the EVM. Gas cannot be bought in the form of a reserve and only serves as an internal accounting-unit of the EVM [5].
For each block of transactions there is a maximum amount of Gas that can be consumed by all instructions combined, known as the *block-Gas limit*. This limit restricts how many transactions can be included in a block. Unlike Bitcoin, which limits the block-size based on the physical size of the transaction-data in bytes, Ethereums block-size depends on the computational-capacity and memory-consumption of the EVM. The current block-Gas-limit is 8 million Gas for each block. This is equivalent to 380 ordinary value-transfer transactions, which by default consume 21,000 Gas [3].

2.5 EVM Bytecode

Solidity source-code is compiled with the Solidity compiler into EVM bytecode. The bytecode consists of a stream of hexadecimal values in big-endian byte-order. EVM bytecode is, therefore, a machine-language that can be interpreted and executed by the EVM. To disassemble the bytecode there are always two bytes of the bytecode used that correspond to a given opcode. That opcode represents a part of an instruction for the EVM. There are opcodes, which form an instruction with an immediately subsequent value. Some opcodes are a valid instruction without a following value (the exact scheme of instructions with opcodes and values is discussed in detail in subsection 2.5.2).
The EVM bytecode can be separated relatively easily into 3 main components: First, the deployment-bytecode as a result of the compilation process. Second, the runtime-bytecode, that is embedded in the deployment-bytecode and there is the meta-data hash at the end of the deployment-bytecode [10]. The exact format and its description will be explained in more detail in chapter 4.

2.5.1 Solidity Compiler

As mentioned before, it is only possible to compile Solidity source-code with a Solidity compiler. For this purpose the *build-target* Solidity-commandline-compiler *solc* is often used. The solc-compiler can generate different outputs from the Solidity source-code. In addition to the regular deployment-bytecode, the compiler can directly generate the runtime-bytecode or the ABI of the smart-contract. Moreover it can generate outputs in the form of *binaries*, *assembly*, *opcodes* or even an *abstract-syntax-tree* to estimate the use of Gas, depending on which tag are set [9].
Using the *–opcode* tag, the raw stream of opcodes is output:

```
$ solc BytecodeDirectory --opcodes SourceFile.sol
```

As an output the *SourceFile.opcodes* is created in the folder *BytecodeDirectory*. This file will display the opcodes in sequence using the optional values in the form of characters. Typically the content of the opcodes file always starts with "*PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO...*" (will be explained in detail in subsection 5.1.1).

More detailed information of the bytecode is output with the *–asm* tag. This outputs a more comprehensive *assembly-items*, containing for example, jump-marks and tags, which are implicitly evaluated by the EVM as storage-addresses in the opcode-stream [9].

```
$ solc BytecodeDirectory --asm SourceFile.sol
```

The command above generates the assembly-items in the *SourceFile.evm* file. This command generates the assembly-items in the *SourceFile.evm* file. The content of this file usually looks like the format as follows:

```
mstore(0x40, 0x60)
  jumpi(tag_1, iszero(callvalue))
  0x0
  dup1
  revert
tag_1:
...
```

The detailed inner-workings of the assembly-items as well as of the opcodes will not be discussed in this thesis. However, more general information about the opcodes can be found in the subsection 2.5.2.

The relevant output for the payload for a contract-creation transaction is created with the *–bin* flag. The output is the opcodes in a binary format, that the EVM can interpret.

```
$ solc BytecodeDirectory --bin SourceFile.sol
```

This command generates the hex-serialized binary bytecode. The output always starts with "*60606040...*" for a valid deployment -bytecode, in most Solidity compiler-versions. The command can also be used to directly output the runtime-bytecode, which usually also starts with the same output in most compiler-versions. This bytecode in the form of hex-serialized binary bytecode can be added as data-payload to a transaction [9].

```
$ solc BytecodeDirectory --bin-runtime SourceFile.sol
```

There is also the possibility to optimize the opcodes in addition to the standard optimization-process using the *–optimize* tag, which will be discussed in great detail in chapter 5 [9].

2.5.2 Structure

This section discusses the structure of the bytecode in more detail. For simplicity reasons is the output of the opcode-stream demonstrated.

Opcodes

An opcode is a certain constant in bytecode with a length of two bytes in case of an EVM bytecode. The 8-Bit *0x60* corresponds for example to the opcode PUSH. The EVM interprets only the binary payload in hexadecimal notation for simplicity, the opcodes are replaced with their identifiers.
In simplified terms every opcode can either add elements to the stack, remove elements from the stack or a combination of both in the context of an operation. The EVM has a collection of opcodes that can be separated into several classes. These include:

- Arithmetic and logical operations (e.g. *ADD, MUL, XOR, NOT*)
- Runtime-environment operations (e.g. *GAS, CALLER, GASPRICE, CODESIZE*)
- Operations to access memory (e.g. *PUSHx, POP, MLOAD, MSTORE, SLOAD*)
- Control-Flow operations (e.g. *STOP, JUMP, JUMPDEST*)
- Logging operations (e.g. *LOGx*)
- System operations (e.g. *CREATE, STATICCALL, INVALID, SELFDESTRUCT*)
- Block-operations (e.g. *BLOCKHASH, DIFFICULTY, GASLIMIT*)

The arithmetic operations are used with the highest stack-elements as operators. For example the opcode *ADD* is adding the upper two elements and places the result back on the stack.
Logic-operations like *LT* (less-than) or *XOR* work bytewise on the operands and use internal flags, which are set in the EVM. Each of the storage-areas can be accessed using different opcodes, depending on the type storage. With *PUHSX* elements of size X can be put on the stack, while they can be removed with *POP*. *MSTORE* and *MLOAD* can be used to access the volatile memory, while *SLOAD* and *SSTORE* can be used to access the permanent storage.
Controlflow operations control the sequence of opcodes and choose the function to jump to. By this the program-sequence are also able to jump from the bytecode-wrappers into the function-bodies (see subsection 4.3.2).
The EVM can determine EVM-internal information of the environment and the block using certain opcodes. For example with the opcode *CALLER* the address of the EOA or contract-account, which initialized the instruction can be loaded. With the *TIMESTAMP* the EVM can get the timestamp of the block that the instruction was included in [1].

Complete Bytecode Instructions

An opcode by itself is in many instances not a complete bytecode-instruction. For some opcodes the EVM expects a value in the bytecode-stream directly followed by an opcode for a valid interpretation of the instruction. The EVM can infer the length of the value in bytes implicitly from the opcode. For example, the opcode *PUSH1* expects a value of one byte [10].

A list of all opcodes of the EVM with the expected values and the operation to execute can be found here[1].

2.6 Yul Intermediate language

Yul, also formerly known as JULIA, is an intermediate language, which can be compiled from Solidity source-code to bytecode of different platforms. This intermediate language is currently still under development and is used in some of the newest Solidity compilers as an experimental feature. At the moment, support for the platforms *EVM 1.0*, *Ethereum-Webassembly* and *EVM 1.5* is anticipated. For inline-assembly (is not discussed in this thesis) in the Solidity programming-language the intermediate language Yul is already used [9].
With Yul different high-level optimizations are possible, which are difficult to implement by the bytecode-optimizer (see chapter 3). In contrast to bytecode, the Yul language is easier to read, the program flow is clearer to understand and the translation into bytecode is more direct. This allows to optimize sections of bytecodes within the Yul intermediate-language. This is difficult to do when using the normal bytecode-optimizer because the context of opcode-instructions in a program is often not easy to grasp [9].
The Yul intermediate-language abstracts from direct memory accesses like *SWAP*, *MSTORE* or *SSTORE* and replaces program flow commands like *JUMPDEST*, *JUMP* and *JUMPI* with "*for*", "*if*" and "*switch*" commands well established in standard high-level programming languages. Furthermore, functional statements are used to demonstrate, which operations are associated with which operands. For example, the bytecode-stream "*7 Y X ADD MUL*" for a stack is rewritten to the functional-command *MUL(ADD(X, Y), 7)* [12].
Yul source-code can also be compiled with the solc Solidity compiler. This is provided by the following command (for the exact specification of the Yul intermediate language, please refer to the Solidity documentation of Yul [9]):

```
$ solc BytecodeDirectory --strict-assembly SourceFile.yul
```

[1]https://github.com/crytic/evm-opcodes

3 Analysis of the Solidity Optimizers

An essential part of the *Solidity-commandline-compiler solc* is the optimizer. More precisely, there is a variety of different optimizers and sub-optimizers, each optimizing for different parameters used in different compilation-stages. The optimization of the Solidity compiler and especially its usage is probably the most misunderstood of all features. This is the reason for taking a closer look at the structure and specific usage of the Solidity compiler. In the context of this thesis references to the source-code of the Solidity compiler are made and optimization processes by debugging the compilation processes are described.
For the basic structure of the optimizer and its usage, we will refer to the latest compiler version (0.7.4). In the context of the bytecode optimizer and particularly in chapter 5 (which deals with changes of the bytecode through optimization) the compiler version 0.5.3 is referred to, because of no interference of the Yul-optimizer. The specific reasons for this decision will be discussed in more detail at the relevant points.
The optimization-process by the Solidity compiler can be separated into two different optimizers: the first one is a bytecode-optimizer, which performs optimizations based on opcode-streams after the general compilation-process and the second is the Yul-optimizer, which optimizes on the level of the Yul-intermediate-language relatively at the beginning of the compilation-process [9].

3.1 Usage of the Optimization in solc

To run all optimization-steps that are available from the compiler, the *–optimize* tag must be appended to the command:

```
$ solc --optimize --bin SourceFile.sol
```

The output of the Solidity *SourceFile.sol* is compiled in bytecode in binary format. By default, this command implicitly passes another value to the compiler, which is the number of expected runs for the the smart-contract. This parameter for the optimizer defines, to what respect the optimizer should optimize the bytecode. The optimizer can either optimize for deployment or for runtime-performance. In the case the optimizer optimizes for deployment-performance, it tries to minimize the bytecode-length as much as possible. As the contract-creation transaction in essence only stores the runtime-bytecode in the permanent-storage once, the shorter the bytecode is, the less Gas will be consumed. Thereby, the main goal for the optimizer is to reduce the number of opcode-commands and values as much as possible. So it doesn't matter how computation-complex or how memory-intensive the opcode-instructions become.
On the other hand you can also optimize for runtime. Here the bytecode is optimized in a way that the execution of the deployed bytecode costs as less Gas as possible. Therefore,

the optimizer tries to simplify the instructions in the bytecode as much as possible. When optimizing for runtime, however, it does not matter how long the deployment bytecode-string gets, in other words, how high the costs of the contract-creation transaction are [9]. This parameter is set manually with the tag *–optimize-runs=n* in the following instruction:

```
$ solc --optimize-runs=200 --bin SourceFile.sol
```

When only the regular *–optimize* tag in the command is set, the default value is 200. This parameter is also called *runs*, which corresponds to the number of expected message-calls of the smart-contract. Therefore, if a low number of *runs* is specified, the bytecode is set to be optimized for deployment. The higher the number of runs is, the potentially longer the bytecode gets, but also the cheaper it becomes to execute the deployed bytecode in terms of Gas costs. To put it in a nutshell, the optimization-process involves a trade-off between optimization of the execution of runtime-bytecode and costs for the contract-creation transaction in the deployment-process. If a smart-contract is expected to be used with high frequency, it is reasonable to initialize the parameter with a high value.
Starting with compiler version *0.6.0*, the Yul-optimizer will be used per default together with the bytecode-optimizer if the compilation-instruction involves the *–optimize* tag. If the optimization of the Yul-optimizer is supposed to be skipped, the compiler must be explicitly instructed to do so with the following command:

```
$ solc --optimize --no-optimize-yul --bin SourceFile.sol
```

Prior to compiler-version *0.6.0*, the Yul-Optimizer was only experimentally included and had to be explicitly declared to use [9].

3.2 Bytecode Optimizer

The bytecode-optimizer works on an assembly basis. A compiler internal data-structure is given as input, which corresponds to a stream of assembly-items (described in more detail in subsection 3.2.2). These assembly-items are reconstructed and returned optimized. In the following, the internals of the optimizer, the recursive optimization of sub-assemblies and the optimization-process in the context of the compilation-process are discussed.

3.2.1 Internals of the Bytecode-Optimizer

The bytecode-optimizer essentially consists of a major loop, in which all optimizers are executed one after another in sequence. The loop is repeated until the sequence of optimizers have made no modifications to the bytecode in the last iteration[1].
The bytecode-optimizer receives parameters for the execution. These parameters contain all essential information that the bytecode-optimizer requires for execution. These settings include a flag whether the *–optimize* tag was set, the number of expected runs, the EVM version and if the bytecode is a deployment-bytecode or a runtime-bytecode[2].

[1]https://github.com/ethereum/solidity/blob/v0.5.3/libevmasm/Assembly.cpp
[2]https://github.com/ethereum/solidity/blob/v0.5.3/libevmasm/Assembly.h

As shown in Figure 3.1, the sub-optimizers *JumpDest-Remover*, *Peephole Optimizer*, *Deduplicator*, *Common-Subexpression-Eliminator* and *Constant Optimizer* are run through in the main loop in the order I referred to. This main loop is always run through, no matter whether the *–optimize* tag is specified or not. So there is always some kind of bytecode-optimization in the compilation-process. Of the five optimizers, the last three (*BlockDeduplicator*, *Common-Subexpression Eliminator*, *Constant ptimizer*) are optional, as indicated in the activity diagram in Figure 3.1. These 3 optimizers are executed in each loop only if the *–optimize* tag has been specified.

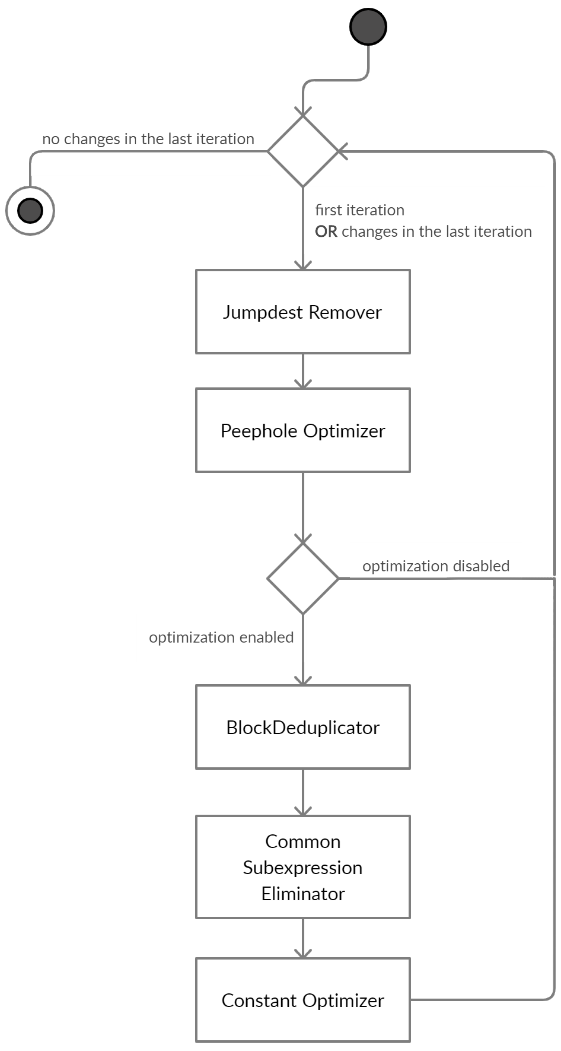


Figure 3.1: Activity diagram of the bytecode optimization-process.

In the subsequent sections, the functionality of the individual optimization-steps will be explained in more detail.

JumpdestRemover

A set of all references is stored and in case of sub-assemblies the references from the parent assemblies are added (tags referenced from outside). These outside-references are passed recursively with each sub-assembly exclusively for the *JumpdestRemover*.
The bytecode stream is matched against the set of references and checked which tags are not referenced by tags in the code. This unreachable dead-code is then removed until the next tag appears in the code[3].

Peephole Optimizer

The so-called peephole optimizer of the Solidity compiler performs the typical peephole optimization-process. This optimization procedure searches for very specific and well defined bytecode-sequences in the bytecode-stream, which can be simplified by other bytecode-sequences that are pre-defined to the optimizer[4].
Hence, the name *peephole*, as only a small fragment of the bytecode-sequence is examined, much like viewing out of a peephole at a front-door. In this process, the context of the bytecode sequences is not taken into consideration and only very specific patterns are searched for in isolation [15].
In particular, the *Peephole-Optimizer* in the Solidity compiler applies a variety of algebraic and logical optimization-techniques. The optimizer bundles operations, streamlines the controlflow, removes dead-code and improves memory-access-efficiency. In the following is a brief explanation of the methods used in the peephole-optimization of the Solidity compiler:

- The optimizer scans whether unreachable code (dead code) is part of the bytecode and removes these sections from the bytecode stream accordingly. In this case all opcodes after an unreachable *JUMP* or *JUMPI* are removed up to the next JUMPDEST. The optimizer also considers code that follows a *STOP, INVALID, SELFDESTRUCT* or *REVERT* as unnecessary. In this case the code is also removed until the following *JUMPDEST*.

- The optimizer performs algebraic simplifications. It searches for algebraic identities, which are mathematical expressions that lead to the same result for all possible inputs. If such an identity is found, this algebraic expression will be replaced by a simpler expression.

- Logical expressions that are linked with *NOT* and *AND* are checked for logical simplification. The optimizer checks if AND-conjunctions are not to simplify and always evaluate to true. Conjunctions of tags with *0xFFFFFFFF* are also removed by the optimizer.

- The controlflow is simplified by immediately placing the target tag on the stack in case of always-true conditions.

[3]https://github.com/ethereum/solidity/blob/v0.5.3/libevmasm/JumpdestRemover.cpp
[4]https://github.com/ethereum/solidity/blob/v0.5.3/libevmasm/PeepholeOptimiser.cpp

- Operations on the stack are also analyzed and optimized. It is checked whether elements are pushed onto the stack, which are then immediately de-stacked again with *POP*. The optimizer also looks for duplicate *SWAP* operations on the stack that can be optimized. Double *PUSH*-operations on the stack can often be optimized as well. The optimizer also checks whether sequences of operations on the stack are to be swapped with *SWAP*, which are potentially commutative and therefore don't need to be swapped on the stack.

The entire assembly stream is scanned with the above mentioned methods and optimized accordingly. The new optimized assembly stream is then returned by the peephole optimizer.

BlockDeduplicator

With the help of the *BlockDedublicator*, duplicate controlflows such as loops or recursive calls are compared with each other and optimized or removed where appropriate. During the optimization-process, virtual tags are added to the assembly-stream, thus, changing the push-tags. These virtual tags identify the current *block* of the controlflow-iteration. If now a *block* is passed through again, which already has a virtual tag that was added by the *BlockDeduplication*, a duplicate recursive call or a duplicate loop was detected.
In order to remove the redundant or unnecessary controlflows, the main-loop of the bytecode optimizer must be run through again in order to delete the duplicate code-blocks. In the process, references are made to the appropriate code-position where the first virtual tag was set, thus, reducing and optimizing the assembly stream[5].

Common Subexpression Eliminator

Probably the most comprehensive and sophisticated part of the bytecode optimizer is the *CommonSubexpressionEliminator*. This optimizer searches among other tasks for components that result in the same output given the same input. These components are bundled into so-called *expression-classes*. If a component cannot be uniquely assigned to an predefined *expression-class*, it is split into simpler expressions and the sub-expressions are assigned to an *expression-class*.
This recursive process involves in addition the memory to be stored in the different memory-sections. The changes of the stored data through the operation-components are noted in a list and associated with the corresponding *expression-classes*. These memory-changes are passed on to the following operation-components. For example, constant instruction-expressions are converted to constant values hard-coded in the bytecode. It is checked whether a set of instructions matches another singular instruction in a given set. It is checked if the instruction expression can be replaced by another instruction expression of the same expression-class [9].
Secondly, with the common *JUMP* and *JUMPI* commands a complete control flow of the whole *assembly-stream* is created, where each operation-block in the control flow is stored together with the memory-manipulation of each operation-block. Based on the described *pre-processing* (creation of the control-flow and the memory-manipulation mapping) among

[5]https://github.com/ethereum/solidity/blob/v0.5.3/libevmasm/BlockDeduplicator.cpp

others the following optimizations are carried out. First, if there are conditional jumps which always evaluate the same, they are converted to jumps. Second, dependencies of the return-values of the last operation-blocks are constructed. Thereby it is checked, if operations in the control-flow that are calculated beforehand in parent-blocks (not needed for the dependency) can be removed. These steps are done recursively for each block [6].

Constant Optimizer

The *Constant Optimizer* is the only optimization process of the bytecode optimizer that includes the expected number of runs of the smart contract in the optimization-process. This is the compilation-parameter that is specified for the *Solidity commandline-compiler solc*. The *Constant-Optimizer* is passed to the parameters whether it is a runtime bytecode or a deployment bytecode, for which EVM version it should be optimized and the expected number of runs. Essentially, constants contained in the bytecode can be rewritten and simplified with a sequence of simple arithmetic operations on the stack. For example, addresses of the originator of a transaction can either be hardcoded in the bytecode as a hexadecimal number or can be calculated dynamically with a number of different operations on the stack. This is described more in detail in chapter 5, where differences between optimized bytecode with non-optimized bytecode are being discussed.
These optimization-methods of rewriting constants by dynamic calculations on the stack to reproduce the constant can be done by the optimizer in any level of complexity. In this context the process of circumscribing values can be used in all possible contexts and intermediate values. This, of course, blows up the bytecode massively, because all possible numbers can be reproduced by diverse operations on the stack. A number of simple operations on the stack with their associated opcodes reserve much more space in the bytecode than to hardcode a simple constant[7].

Recursion over Sub-Assemblies

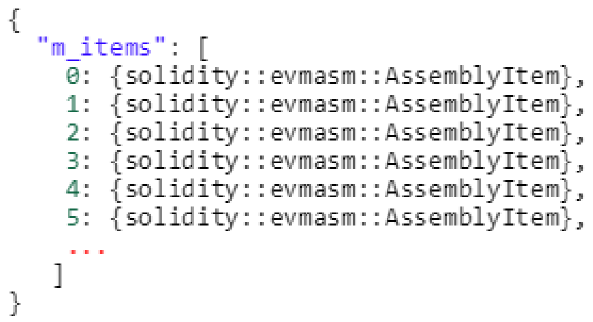
The actual optimization-process of the bytecode optimizer works recursively over the assembly-stream. For each sub-assembly of the assembly-stream, the entire bytecode-optimizer is performed again beginning from the very start. Sub-assemblies are opcode-streams, that cannot be processed by the controlflow of the native bytecode. For example, these can be assemblies such as the runtime-bytecode, which is embedded in the deployment-bytecode (see chapter 4) and is only written in permanent storage. In this case runtime-bytecode as sub-assembly is never executed in the direct controlflow of the opcode-stream of the outer deployment-bytecode. However, the runtime-bytecode stream is embedded in the deployment-bytecode and is fully optimized as a sub-assembly. This subtle feature enables the runtime-bytecode to be optimized as a sub-assembly of the deployment bytecode when compiling deployment-bytecode. In case of compiling bytecode for a contract-creation transaction, the embedded runtime-bytecode is also optimized and can be called after the deployment-process efficiently.

[6]https://github.com/ethereum/solidity/blob/v0.5.3/libevmasm/CommonSubexpressionEliminator.cpp
[7]https://github.com/ethereum/solidity/blob/v0.5.3/libevmasm/ConstantOptimiser.cpp

Another example of a sub-assembly is the creation of a new smart-contract using the bytecode of an already existing contract. The bytecode of the new contract, embedded in the already deployed bytecode, is also a sub-assembly, which is also optimized recursively in the original compilation-process.

3.2.2 Optimization in the Compilation Process

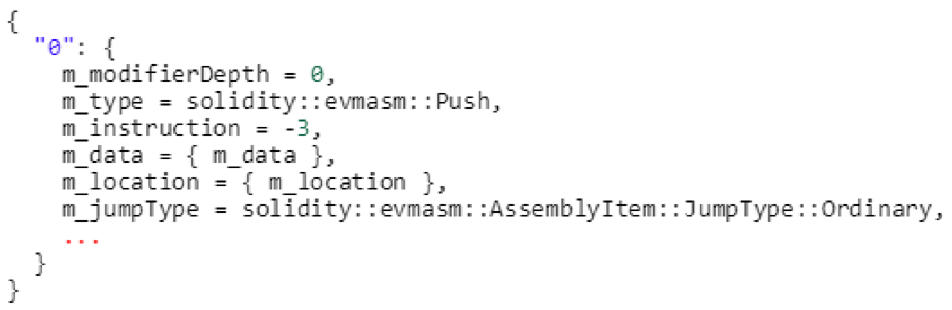
Bytecode-optimization takes place during the compilation process as one of the last steps. After compiling the bytecode and appending the meta-data to the smart-contract (see chapter 4), the settings for the optimizer are initialized and then the optimization method is called.



```
{
  "m_items": [
    0: {solidity::evmasm::AssemblyItem},
    1: {solidity::evmasm::AssemblyItem},
    2: {solidity::evmasm::AssemblyItem},
    3: {solidity::evmasm::AssemblyItem},
    4: {solidity::evmasm::AssemblyItem},
    5: {solidity::evmasm::AssemblyItem},
    ...
   ]
}
```

Figure 3.2: AssemblyItems data-structure in the compilation-process.

At this time of the compilation-process the bytecode is still in the form of an Assembly-Item data-structure (as seen in Figure 3.3). After the bytecode-optimization the data-structure is transformed into the specified output like assembly, opcodes or binary. The exact description of the data-structure in memory at the time would go beyond the scope of this work. However, parts of the data structure which are necessary for the optimization and which are processed are to be described.
For the optimization process, a list of assembly-items *(m_items)* (as seen in Figure 3.2) containing all assembly-elements is relevant. These assembly-items are divided into different types *(m_type)* such as *operations*, *push elements*, *tags* or *push data*. All relevant information such as the values *(m_data)* associated with the instructions or the labels of the tags are located as attributes with the assembly elements in a map. These attributes are not discussed in detail due to the limited scope of the thesis and the complexity. The exact instruction of a type *(m_instructions)* is an enumeration, which is initialized at runtime with an always different integer value. This is an inherent property of the programming language C++, that the values for the instruction enumeration correspond to a different value depending on the runtime.



```
{
  "0": {
    m_modifierDepth = 0,
    m_type = solidity::evmasm::Push,
    m_instruction = -3,
    m_data = { m_data },
    m_location = { m_location },
    m_jumpType = solidity::evmasm::AssemblyItem::JumpType::Ordinary,
    ...
  }
}
```

Figure 3.3: AssemblyItem data-structure in the compilation-process.

3.3 Yul Optimizer

In contrast to the bytecode optimizer which is described here in more detail, the *Yul-optimizer* does not work with assembly, but with the Yul intermediate-language.
By default, the *Yul-optimizer* uses a number of predefined optimization-processes. This set of optimization-procedures of the Yul-optimizer can be overridden with the following command:

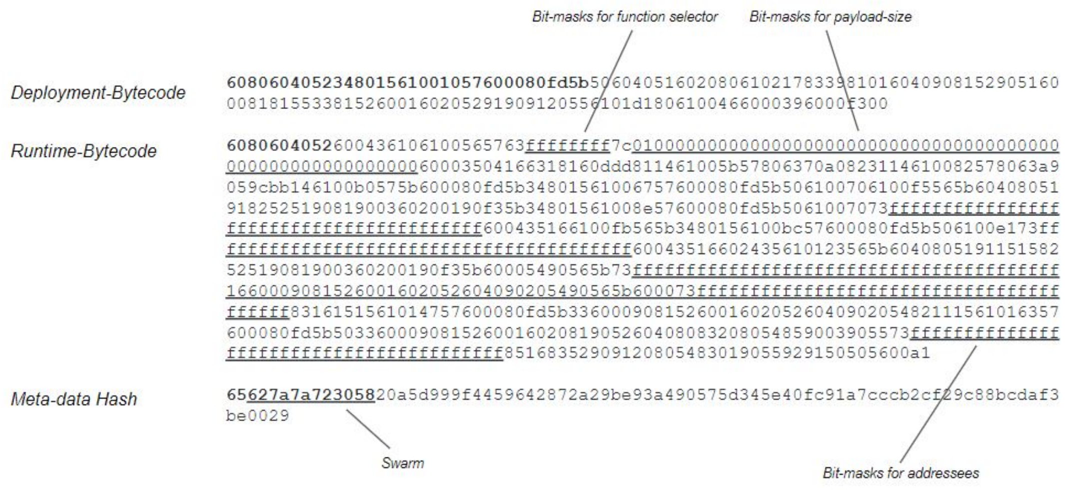
```
$ solc --optimize --yul-optimizations dhfoD[xarrscLMcCTU]uljmul
```

There is a variety of sub-optimizers such as the *BlockFlattener*, *CircularReferencesPruner*, *CommonSubexpressionEliminator*, *ConditionalSimplifier*, *ControlFlowSimplifier*, *DeadCodeEliminator* and many others. In the above written command each character represents a special optimization-process in the Yul optimizer[8].
Unlike the bytecode-optimizer, the order of the optimization methods plays a major role here, which is why you can specify them by indicating characters in the command. Some optimization-processes are also required in order to perform other optimization-procedures. With the square brackets you can specify to repeat the bracketed optimization-processes until no more optimizations are done. This follows the same concept as the bytecode optimizer. For detailed background information on the individual optimizers and a complete list with the respective characters, please refer to the following documentation [9].

[8]https://github.com/ethereum/solidity/tree/v0.5.3/libyul/

4 Bytecode Structure

The bytecode consists of byte-sized hexadecimal values in big-endian notation (network byte-order). The bytecode is a more compact way of setting machine-instructions than assembly-code. In the bytecode there are only opcodes with their values declared, instead of additional unused data for the EVM like the tag-identifiers in assembly-annotation. Which type of bytecode the solidity compiler outputs, has to be specified as a parameter in the compilation-instruction. By default this is deployment bytecode. The use of the deployment-bytecode in a contract-creation transaction as well as the interpretation of runtime-bytecode has already been described in detail in the previous chapters. In the following, the structure of the individual bytecodes as well as the differences in structure will be discussed.
The deployment-bytecode, or rather the actual bytecode for the deployment process can be divided into three essential parts, as shown in the Figure 4.1 below. This bytecode originated from a relatively basic Solidity source-code, which can be accessed from the following source[1]. However, the particular functionality of the smart-contract is not of importance in this context. The 3 parts are the *deployment bytecode*, the *embedded runtime bytecode* and the attached *meta-data hash*. Strictly speaking, all 3 bytecode-parts combined are the deployment-bytecode, but the actual executed bytecodes-sections in the deployment-process and in runtime can be differentiated.



```
608060405234801561001057600080fd5b5060405160208061021783398101604090815290516 0
00818155338152600160205291909120556101d1806100466000396000f300

6080604052600436106100565763ffffffff7c0100000000000000000000000000000000000000
000000000000000000006000350416631 8160ddd811461005b57806370a0823114610082578063a9
059cbb146100b0575b600080fd5b3480156100675760008 0fd5b506100706100f5565b60408051
918252519081900360200190f35b34801561008e57600080fd5b5061007073ffffffffffffffff
ffffffffffffffffffffffff600435166100fb565b3480156100bc57600080fd5b506100e173ff
ffffffffffffffffffffffffffffffffffffff60043516602435610123565b6040805191151582
525190819003602001 90f35b60005490565b73ffffffffffffffffffffffffffffffffffffffff
16600090815260016020526040902054 90565b600073ffffffffffffffffffffffffffffffffff
ffffff8316151561014757600080fd5b33600090815260016020526040902054821115610163 57
600080fd5b503360009081526001602081905260408083208054859003905573ffffffffffffff
ffffffffffffffffffffffffff85168352909120805483019055929150505600a1

65627a7a7230582 0a5d999f4459642872a29be93a490575d345e40fc91a7cccb2cf29c88bcdaf3
be0029
```

Figure 4.1: Bytecode-segments of a compiled smart-contract.

[1]https://github.com/jonasgebele/ba_bytecodes

4.1 Identical Bytecode EVM-Initialization

It can be observed that deployment bytecode as well as runtime bytecode always prefixes the opcodes-stream *0x6080 6040 52 34 80 15 610010 57 6000 80 fd 5b* (as indicated in Figure 4.1). Precise values of the bytecode-streams may vary depending on the compiler version. Here the Solidity Compiler *version 0.5.3* is examined. The significance of this bytecode-sequence of numbers shall be explained in the following.
The first two instructions *0x60806040* use *PUSH1 (0x60)* to move the byte values *0x80* and *0x40* to the stack. As the EVM has a stack-based architecture and therefore no registers, it is only able to execute instructions with values that are stored on the stack. In order to access them with the following instruction, the values have to be pushed on the stack first. The following operation *MSTORE (0x52)* uses these two values from the stack in order to store into the volatile memory. The first argument specifies the address *(0x40)* of the second argument as the value to store to volatile memory. Translated into Yul this would correspond to *mstore(0x40, 0x80)* [1].
This memory-space is allocated in volatile memory for the *free-memory-pointer*, which Solidity usually abstracts in the source code. The space between *0x00* and *0x40* in the volatile memory for the EVM is reserved for special hashes (which are not discussed in this thesis for reasons of complexity and the scope [1].
With *CALLVALUE (0x34)* the number of ether of the message-call is put on the stack. *CALLVALUE* is an environment instruction, which is correctly interpreted and executed by EVM. This stack value is duplicated with *DUP1 (0x80)*. *ISZERO (0x15)* places a 1 on the stack if the topmost element on the stack is a 0, in other words, if the number of ether of the message call is 0. Then *PUSH2* puts 2 bytes with the value *0x0010* on the stack to set the destination for the following *JUMPI* command.
This command jumps to *JUMPDEST* (directly after *REVERT*) and starts with the constructor in the specific deployment-bytecode section, if a positive ether-value is added to the message-call. If there are no ethers attached to the message-call then *REVERT (0xFD)* is executed and the EVM terminates with an error code which is pushed onto the stack beforehand. This bytecode-logic would correspond to something like the following code written in Solidity:

```
if(msg.value != 0) revert();
```

This piece of code was not explicitly used in the Solidity code, but was implicitly added by the Solidity compiler. This piece of code was not explicitly used in the Solidity code, but was implicitly added by the Solidity compiler. Since the constructor of the smart-contract was not defined as *payable*, Solidity adds the following piece of code to make sure that the constructor does not receive ether [16].
This previous section is not specific to deployment or runtime-bytecode and has, therefore, been discussed in general terms. From this point on, the individual bytecodes begin to differ. Therefore, the differences in the structure and setup of both bytecodes will now be discussed in more detail.

4.2 Deployment Bytecode

The deployment code contains the creation-code and the execution of the constructor to initialize the smart-contract account. As already mentioned in this chapter, this bytecode-section is only executed once by the EVM for the deployment of the runtime-bytecode. When the creation code is executed, a copy of the runtime bytecode is returned, which is the actual smart-contract bytecode.
At the very beginning of the specific deployment bytecode the parameters for the constructor are loaded into the volatile memory. The parameters are located at the end of the general bytecode. For this purpose the environment-instruction *CODECOPY (0x39)* is executed. As parameters for this instruction, the number of bytes to be copied, the position of the instructions to be copied of bytecode and the position in the volatile memory where you want to copy to are pushed onto the stack.
Moreover, as already mentioned in the last chapter, the constructor for initializing the smart-contract during the deployment process is located in the deployment bytecode. It is located in the bytecode immediately after storing the parameters. As this constructor is highly individual for each solidity code, we described it in a very abstract way. Generally speaking it can be said that for the initialization with the constructor, values are usually stored in the permanent storage using memory-access operations like *SSTORE*, in order to define values in the smart-contract account with the constructor.
After executing the constructor, a *CODECOPY* is used this time to store the runtime bytecode with an individual length from an individual position in the bytecode code to the fixed position *0x0046* in the volatile-memory.
The end of the deployment bytecode is marked with a *PUSH1 (0x60)* of the value 0x00 onto the stack, followed by a normal *RETURN (0xF3)* opcode. Finally, depending on the compiler version, a *STOP 0x00* or an invalid expression such as *0xFE* as in compiler *version 0.5.0* can be found [16].

This general structure of the deployment bytecode is always similar and does not differ fundamentally depending on different smart-contracts or different compiler-versions. The only differences are specific sections on certain parts of the bytecode like the initialisation of the constructor or the length of the bytecode to be copied. This general structure is especially relevant if we want to find out differences by optimization in the compilation process of EVM-bytecode in the following chapter.

4.3 Runtime Bytecode

The runtime-bytecode is much more individual than the deployment bytecode, but you can separate it into different sections in the same way. In the following, each section will be discussed and the basic functionality will be explained. This background information will be used later when differences in the bytecode will be discussed in the next chapter, which are due to compiler optimization.

The runtime-bytecode is the bytecode that is executed each time the smart-contract is called with a message -all. Incoming transactions are interpreted the same way as with the deployment-bytecode but transactions are not sent to the *zero-address*. Thus, the runtime bytecode starts in the same way as the deployment bytecode with the *Identical Bytecode EVM-Initialization* [16], as described above.

4.3.1 Runtime-Dispatcher

At the beginning of the bytecode you find the environment-instruction *CALLDATA*, which places information about the message-call *(msg.data)* on the stack (for detailed information about the send data of a message-call refer to subsection 2.2.2). If the received data-length is only 4 bytes, these 4 bytes get interpreted as the function-selector. The message call passes the *Keccak256* hash of the function-name with parameters to identify the function to be called in the bytecode. The first 4 bytes of the hash value are used to identify the function in this context. These function signatures are compared with the bytecode using a dispatcher in the runtime bytecode. The dispatcher compares the function-signature with the function-selectors and jumps to the corresponding function in the bytecode if one matches. Therefore, this section is called dispatcher, because it corresponds exactly to such a mechanism. If no signature matches, a return value is pushed onto the stack and the bytecode ends with a *REVERT* (in the case of a Fall-Back function, it jumps to this function instead) [1].
To check if the message-call data is only 4 bytes the bytecode always pushes a 4 onto the stack and checks with *CALLDATASIZE* the size of the data on the stack. If the *CALLDATA* opcode puts the whole 32-Bytes of message-call data on the stack, byte-masks are used. To interpret this data we can often find in the bytecode of the runtime-dispatcher address-masks with 4 bytes of *0xffffffffffff* and a 29 byte-mask with a relatively small value (one 1 and the rest 0) to extract the function selectors from the 32 bytes. This method of extracting the function-selectors hash using bit-masks is also discussed in the next chapter in the context of optimization and resulting changes in the bitmask. In general the logical *AND* opcode is used to extract the 4 bytes of the function identifier and the other mask is used to reduce the 32 bytes to 4 bytes with a division containing the *DIV* opcode. This reduces the 32 bytes of message-call data with parameters to the 4 byte function selector [16].

These 4 bytes are then compared with hard coded identifiers via *EQ* and, in case of a match, are the functions in the bytecode are accessed via *JUMPI*.

4.3.2 Function Wrappers

When the *Runtime-Dispatcher* jumps to a function, it is not executed immediately. In the beginning, the EVM must be prepared by the bytecode of the function so that it can be executed.
If the function is declared not-payable, the *Bytecode Initializer* is added, as discussed in section 4.1. If necessary, additional values on the stack get cleaned up. At this point values from the permanent-memory must also be loaded onto the stack if the smart-contract needs to

access account-values in the function.
In some cases further data like addresses from the sender need to be loaded, as well. These are also processed in the wrapper in the same way as in the Function Selector with bit-masks and pushed on the stack. Additionally, here you can find optimizations at the bytecode, as described above. These bit-masks can be recognised with *0xffffffffffffffffffffffffffffffffffffffff* constants in the bytecode [10].

The function wrapper embeds the function body. After the actual Function Bodie the Return Type is defined and placed below the actual result of the Function Body and gets returned with *RETURN* [16].

4.3.3 Function Bodies

The *Function Body* is the actual function implemented in Solidity. To execute the function Body, the EVM needs all parameters and values from the permanent storage already on the stack in order to use them (as this code and its functionality is highly individual from the implemented source code, it will not be discussed in detail in this context).

4.4 Metadata Hash

The *Meta-data Hash* is part of the bytecode, but is not executed by the EVM. The hash is also in the bytecode just behind the *STOP* opcode and is not jumped to with a *JUMPDEST*. It is simply appended to the end of the bytecode, just like the parameters of the deployment bytecode. This hash contains meta-data about the source code, the used compiler and the settings during the compilation-process.
The purpose of this hash is to store the meta-data in a system called *Swarm*, a decentralized storage-system. This allows users to independently compile the source code with the specified compiler and verify the bytecode on the blockchain. Thus, not only transactions and instructions of smart-contracts can be verified, but also the bytecode of the Smart Contracts themselves [16].
At first the logging-Opcode *LOG1* is processed and then the following 6 bytes are pushed onto the stack. This is the sequence *0x62 7a 7a 72 30 58*. The six characters are the ASCII notation for "*bzzr0:*", which is a reference to the Swarm-system leading to the informal term Swarm-Hash. After the sequence comes a *SHA3* opcode and followed by the actual metadata-hash, which has a length of 41 bytes. The hash is encoded with Concise Binary Object Representation [9]. This swarm hash can also be created with a command in an external file with the solidity commandline compiler solc if this is specified with a flag in the compilation instruction [1].

5 Analysis of optimized Bytecodes

The following section explores how the additional optimization with the use of the *–optimize* tag by the Solidity compiler has an effect on the bytecode-level. Special focus will be put on the former chapter in order to work out specific sections of the bytecode, where changes can be detected straight forward. The optimization does not refer to the standard bytecode-optimization which is done per default by the Solidity compiler but to the additional optimization-steps by setting the *–optimize-runs=200* tag in the compiler-instruction.
Various Solidity source-files are used as a basis for these findings.These are relatively compact in order to keep the bytecode as manageable as possible and still cover as many Solidity features as possible. These Solidity source-files as well as the optimized and non-optimized bytecodes can be found at the github-source[1]. The compiler used was the *Solidity-commandline-compiler solc version 0.5.3*.
All gathered insights refer to the explicit exclusion of the Yul-optimizer. The Yul-optimizer must be explicitly enabled in older compiler versions (e.g. *0.5.9*), but explicitly excluded in newer compiler versions (from version *0.6.2*) in order to replicate the information. The purpose of this chapter is to show structural changes that occur generally during optimization and are not specific to individual smart contracts.

5.1 Differences in sections of the bytecode-structure

Now we will discuss in more detail how additional optimization using the *–optimize* tag in the compiler-instruction impacts the structure of the bytecodes in the different sections as well as which parts change and how the changes impact the bytecode. Each section is preceded by a short conclusion, followed by a more detailed explanation of the changes caused by the optimization-process.

5.1.1 Differences in the EVM-Initialization Bytecode-section

No changes in the free memory-pointer and non-payable constructor-check - The initialization of the volatile memory and the allocation of memory for the free memory pointer for the EVM remain unmodified. The Identical Bytecode EVM-Initialization, which checks whether ether are sent to the constructor, cannot be optimized or simplified by the bytecode-optimizer since not checking could lead to a runtime-error in the EVM.

[1]https://github.com/jonasgebele/ba_bytecodes

Minor changes in exit-condition of constructor-check - With the exception of a small constant change, the bytecode section of the exit-condition of the constructor-check is unmodified. A different address of a *JUMPDEST* destination has to be pushed onto the stack, which will be jumped to in case of an error. The reason for this modification is obviously because the rest of the code and the number of instructions changes through the optimization-modifications and, therefore, also the position of the *JUMPDEST*. This marker is always placed on the stack as the ninth instruction with *PUSH2*.

In summary, the bytecode initialization of the deployment-bytecode as well as the runtime-bytecode is substantially unmodified, with the exception of the jump-tag of the exit condition (if ether are sent to a non-payable constructor). The same conclusion about the check of non-payable constructor can of course be applied to non-payable functions. In this case the optimizer does not make any modifications as well.

5.1.2 Differences in the Deployment Bytecode

In the following, the deployment-bytecode is analyzed with respect to the sections of the constructor-execution, parameter storage and runtime byte code and structural changes are shown.
Minor changes in storing constructor-parameters - If parameters in the bytecode are passed for the constructor, they also have to be copied from the end of the bytecode into the volatile memory. The only difference in this context is the *CODECOPY* opcode with the argument of the position, from where the parameters are positioned. As the rest of the bytecode is heavily modified by the optimization, the position of the parameters at the end of the bytecode is obviously different.

Major changes in the constructor body - As the following execution of the constructor is highly individual for the source code of the smart contract, no structural changes can be observed here. Nevertheless it is to point out that the optimizer reduces extremely cost-intensive opcodes like memory accesses to the permanent memory to a minimum. As especially during the execution of the constructor many values of the smart-contract account need to be set, there are many accesses to the permanent memory in this section. Related to the analysed examples the optimizer reduced the number of calls to *SSTORE* opcodes from 2 down to 1.

The optimization of memory accesses in general is discussed in subsection 2.4.1. In the context of this section we take a closer look at structural changes.

Minor changes in storing the runtime-bytecode - Just like the process of storing parameters in the *volatile memory*, the optimization causes hardly any changes when storing the runtime bytecode with *CODECOPY*. Only the parameter, which indicates the position in the bytecode from which the copy is to start, is changed.

There are no changes when the deployment bytecode is terminated. The return value is still placed on the stack and the program is terminated with *STOP*.

5.1.3 Differences in the Runtime Dispatcher

Instruction-Iterations in the Runtime-Dispatcher - The runtime dispatcher shows a rearrangement of the instructions in the analyzed bytecodes by the optimizer. This is presumably due to the restructuring of the optimizer. Small optimizations related to the gas-costs can be observed, but are mainly due to a change in the order of the PUSH operations on the stack, which can save *SWAP* operations. Changing the order of pushing parameters onto the stack therefore saves gas during execution because of easier and more intelligent access to these values.

No optimization in the extracting data from CALLDATALOAD - No optimization has been found to extract the function identifier from the message-call data *(msg.data)*. *DIV* and *AND* are still used to reduce the 32-byte to the 4 bytes and extract the identifier.

5.1.4 Differences Function Wrappers

No structural optimization - In the function wrappers relatively few structural changes by the optimization-process could be detected. Since the volatile memory in the wrappers has to be accessed frequently, the optimizer tries to minimize the access to this storage-section. However, there are relatively few optimization capabilities because the wrappers provide the parameters and return types for the EVM. Since parameters and return types cannot be optimized off and are necessary for the execution of the function-bodies, they are structurally impossible to optimize. However, as in the constructor body, stack operations are optimized in stack operations, which will be discussed in a later section.

5.1.5 Differences in Function Bodies

No structural optimization - In the Function Body no concrete change behavior patterns could be detected that relates to the structure of the Function Body and goes beyond general operation as described in subsection 4.3.3.

5.2 Differences in general Bytecode-Instructions

While in the optimizations described above the general structure of the bytecode was considered, now general optimizations of opcode-streams are examined, independent of the context of the bytecode-section.
As the analysis of the optimizer of the Solidity compiler showed, only the *BlockDeduplicator, Common-Subexpression-Eliminator* and the *Constant-Optimizer* are used for the additional optimization during a compile process.
The *BlockDeduplicator* detects and combines redundant code sections. These code sections result from duplicate loop constructions with similar or reusable exit-conditions. This is

especially important for the optimization of the deployment procedure with a low number of runs, but according to the implementation of the Solidity compiler it is actually independent of the run parameter, which is quite surprising.
The *Common-Subexpression-Eliminator* looks for certain bytecode patterns which it can simplify. In a nutshell, instruction-expressions are recursively divided into certain instruction-classes (described in more detail in chapter 3). These instruction-expressions are searched for certain-patterns. The *Common-Subexpression-Eliminator* converts instruction-expressions with always-constant values into constants. Expressions are simplified. Expressions are replaced by other expressions within an expression class according to gas costs. Furthermore it is checked if certain instruction-expressions can be replaced by a simple opcode instruction.
The *Constant-Optimizer* replaces defined constants in the bytecode with a dynamic calculation on the stack. Any number of intermediate results can be reproduced on the stack by simple operations. The *Constant-Optimizer* is also the only optimizer with an influence of the number-runs parameter during the compile process.
More specifically, the *Constant Optimizer* is used for constants hard-coded in bytecode, such as address masks. If, for example, special addresses of the sender of the instruction are used in the bytecode or in the function dispatcher to extract the function-identifiers from the data-payload, these are hard-coded in the bytecode in the un-optimized state. The masks are pushed onto the stack and applied by logical operations like *AND* with an address or another payload to have the value with the correct number of bytes on the stack, as demonstrated in the following bytecode-segment.

```
CALLER
PUSH20 0xffffffffffffffffffffffffffffffffffffffff
AND
```

However, such address masks can also be created dynamically on the stack by a sequence of instructions like the following. As an example, to get the address of the sender with the right size on the stack, can be programmed in Solidity using *msg.sender* and the constant-optimizer could convert it to the following bytecode.

```
CALLER
PUSH1 0x01
PUSH1 0xA0
PUSH1 0x02
EXP
SUB
AND
```

The result is calculated from the base 2 with the factor 160, then subtracting 1 to get the same constant and finally do the same logical AND operation to apply the bitmask. This bitmask is logically combined with the payload of the caller (which is the address of the sender of the instruction) to get the address with the correct size on the stack.
These bitmasks occur in all possible situations, for example in the function dispatcher or in the function wrappers, if you expect parameters or return values of a function in a certain

type.
In general we can conclude that many long constants with all bits set in the bytecode indicate missing bytecode-optimization, because they should always be created by operations virtually on the stack. If the bytecode has been optimized extensively for deployment, this is even more the case, because such a constant takes up much more memory in the bytecode than a few instructions [1].

6 Implementation of a Re-Optimizer

The results achieved provide an indication of how the bytecode-optimizer is structured and with which bytecode-sections and patterns in the bytecode it is possible to differentiate optimized bytecode from an un-optimized bytecode. To be able to distinguish whether a bytecode of a smart-contract is compiled with the *–optimize* tag is difficult. With the results so far we can only estimate probabilities to what extent two bytecodes are based on the same source code. However, in this context it is not possible to differentiate deterministically, especially not in a large scope where all smart-contracts on the Ethereum blockchain are analyzed for optimization. When comparing two specific bytecodes, it would be feasible to compare the control flow graph with the Gas costs of the instructions, which would provide a suitable estimate for at least some bytecodes.
Another approach would be to re-optimize the analysed bytecode. If the re-optimized bytecode would be different from the original bytecode, this would indicate that the optimization was missing in the compilation-process. At the same time, this approach can be used to detect redundancy of stored smart contracts on the Ethereum blockchain due to missing optimization. Redundancy can be obtained when the re-optimized bytecode is found again already existing on the Ethereum blockchain.
This requires further bytecode-optimization of the existing bytecodes stored on the Ethereum blockchain. How to implement such a re-optimizer of existing bytecode on the Ethereum blockchain is discussed in the next section, where possible technical and conceptual challenges will then be addressed in more detail such as an optimizer goes beyond the scope of this bachelor thesis, especially due to the challenges that are encountered. In this chapter we will only deal with a conceptual design of such a re-optimizer. A concrete implementation would be part of the future work.

6.1 Design of the re-optimizer

The bytecode-optimizer is part of the Solidity compiler. It works with the previously introduced data-structure Assembly-Items *(m_items)*, as described in subsection 3.2.2. The bytecode-optimiser restructures these assembly-items in the optimization-process and returns a new list of assembly-items as a result.
At this point, if we want to simulate the optimization with the *–optimize* tag, it is preferable to only run the optional optimizers over again instead of the entire optimization-process of the bytecode-optimizer. The required optimizers for this purpose include the *Common-Subexpression-Eliminator*, the *Block-Deduplicator* and the *Constant-Optimizer*. It is also recommended to separately run the runtime-bytecode and the relevant deployment-bytecode section

in the case of optimizing a deployment-bytecode. Moreover, the meta-data hash must be removed because it is appended during the compilation process after the optimization.
This is where a parser needs to be developed to convert the binary hex-bytecode into the required AssemblyItem data-structure with which the Solidity compiler operates. This is where a parser must be developed to convert the binary hex bytecode into the required AssemblyItem data structure. It is important to consider that the AssemblyItem data structure represents assembly code and therefore contains considerably more detailed information than regular EVM-bytecode. Firstly, unknown data such as identifiers from *jump-labels* must be abstracted and named. Secondly, many attributes of the AssemblyItem data structure will be uninitialized, as the entire compilation process preceding the optimization is skipped since we start with the optimization-process. Essentially only the stated AssemblyItems data structure is relevant for the bytecode-optimizer, which is a list of different assembly-items. For these assembly-items the optimizer needs the type of the instruction *(m_type)* as well as the particular instruction *(m_instruction)*. The optional data payload of the instruction *(m_data)* is initialized differently depending on the opcode (more information about the data structures is provided in subsection 3.2.2).
The actual optimization can be performed by the current bytecode-optimizer. This optimizer does not have to be re-implemented but can be re-used entirely. Only the standard bytecode-optimizers (*JumpDest Remover, Peephole Optimizer*), which are executed also if the *–optimized* tag is not set, can be removed without any problems. It is important to make sure that the optimizer settings (see chapter 3) are correctly initialized so that the needed optimizers are performed and the expected number of runs is correctly initiated. A default value of 200 will be assumed. The respective optimizers only needed the AssemblyItems data structure as input (except for the *Constant Optimizer*).
At the end it must be considered that the compiler has to be terminated explicitly after the modified optimization because the further compiling process is not possible due to the missing initialization of different values. Certain functions of bytecode output are already implemented in the Solidity compiler and can be reused for the modified bytecode

6.2 Challenges

The potential implementation of such a re-optimizer faces several challenges of technical and conceptual nature. These make the implementation of a re-optimizer particularly difficult and raise questions about the fundamental feasibility and correctness of the results. The bytecodes to be optimized have to be restricted in order to achieve a re-optimization with the same bytecode of a originally optimized bytecode. See the following section for more details.

6.2.1 Conceptual Challenges

One conceptual problem that might occur is that even with a perfectly correct optimization, the exact same bytecode might not be produced by a re-optimizer. For a usual optimization with *–optimize* tag at compile-time, the input of the three optional optimizers is a non-optimized

bytecode at each iteration of the main-loop. In contrast to that, a re-optimizer provides 3 optional optimizers with finish-optimized bytecode (from the standard-optimizers), because the partially optimized bytecode already has many iterations of optimization done by the default compiler.
As a result, the re-optimizer has a different assembly input at each iteration than the usual optimizer, because they work with bytecodes optimized to different degree. For this reason both optimizers do not necessarily come to the exact same bytecode as a result. These presumably small differences can be qualified by metrics like the *Levenshtein distance* with a margin of error.
Another challenge is the frequent changes made to the Solidity Compiler with many different Solidity Compiler versions. Although the relevant bytecode optimizer has remained fairly untouched in its basic structure, there are several smaller bug-fixes and changes. For example, version *0.5.9* introduced a rule to simplify *SHL/SHR* shift-instruction combinations and version *0.5.10* implemented the simplification-rule from *SUB(0,X)* to *NOT(X)*. In principle, it is, therefore, necessary to restrict the dataset of the analyzed bytecodes to certain compiler versions or to implement multiple re-optimizers, depending on which compiler the bytecode was originally compiled with. A possible solution would be to only consider smart-contracts verified on *Swarm*, because in this way we are able to check the compiler-version of the used compiler. However, in this case we would not need a re-optimizer, because we could read the instruction with which the bytecode was compiled.
Another restriction of the bytecodes to be optimized with the re-optimizer is to consider only compiler versions that don't use the Yul-Optimizer by default in the optimization process. The Yul-Optimizer, which optimizes on the Yul intermediate-language between Solidity source-code and bytecode, has a relatively large influence on the generated bytecode. This Yul-Optimizer is used in the several compiler versions in different compilation-instructions. In the first compiler versions it had to be specified in the compiler instruction that the Yul-Optimizer should be executed. Later the Yul-Optimizer was executed whenever the *–optimize* tag was set [9].
Re-optimization of bytecode, so that redundancy on the Ethereum blockchain can be detected, obviously only works if the Yul-Optimizer was used for both bytecodes that are being compared or if the Yul-Optimizer was not used at all. For this reason, bytecodes which were compiled with one of the latest Solidity compilers should therefore be ignored, because the re-optimizer can only optimize bytecodes and there cannot optimize regarding the Yul intermediate-language.
Moreover, it must be assumed that the source-code has been compiled and optimized with the default-value for the expected number of runs. Although this is not always the case, many development environments integrate the Solidity commandline compiler, such as *Remix*, and therefore automatically initialize the compilation-instruction with the value 200.

6.2.2 Technical Challenges

As already mentioned in the Conceptual Challenges the bytecode contains much less information than an assembly-code. Thereby it is only possible to translate the bytecode into assembly-code by making certain assumptions like naming the jump labels. But there are also more problematic cases than the names of markers. While opcodes in bytecode have a relative general functionality, there are a quantity of assembler-commands in assembly-code, which are translated to several opcode-instructions. This assignment of assembler-instructions to opcode-instructions is not always unique and often only identifiable from the context of the opcode-instructions. For example the assembler-instruction *pushString* is translatable by a *PUSHX* opcode. Nevertheless a *PUSHX* opcode in combination with other opcode instructions can represent not only a *pushString* assembly-instruction, but also other types of push assembly-instructions. This example shows that the assignment of opcodes to assembler-instructions in some cases depends on the context of usage, which makes parsing the bytecode to AssemblyItems quite challenging.

7 Conclusion and Future Work

In this last chapter the results of the bachelor thesis are summarized and provide an outlook for further work, especially considering a specific implementation of the re-optimizer.

7.1 Conclusion

At the beginning of this thesis we started with a brief overview of the Ethereum foundations. Besides a short introduction to Ethereum fundamentals, we covered the components of an Ethereum-transaction as well as the transfer of data by means of transactions. Afterwards, we discussed the development of smart-contracts, how the EVM handles them and how they are implemented using high-level programming languages like Solidity. We then introduced the Ethereum Virtual Machine first from a theoretical point of view with its global singleton state and state transitions, followed by the operational aspects of the different memory-sections and the concept of Gas. Then we discussed how the EVM bytecode is generated with the Solidity compiler and how it is structured and evaluated by the EVM. Last but not least provided a basic overview of the Yul intermediate-language that operates between the Solidity source-code and the EVM bytecode.
Afterwards, we began with a detailed analysis of the different optimizers of the Solidity compiler with references directly from the implementation of the source-code of the compiler. This chapter aimed to answer research question 1 as well as providing a first detailed and comprehensive documentation of the functionality and the structure of the different bytecode-optimizers. We explained the effect of compiler instructions and parameters like the expected number of executions of a smart-contract regarding the optimization in the compilation-process. Another area of research was the compilation-context in which the optimization is performed and how the bytecode is represented internally in the compiler. Before providing answers to the following research questions, in the next chapter we examine the structure of the EVM-bytecode in detail. By separating the bytecode into different bytecode-segments and outlining their functionality, changes through optimization discussed in the next chapter can be better understood. In doing so, we explained the difference between runtime-bytecode and deployment-bytecode. We also described the execution of the two bytecode types as well as the individual aspects and operations for the EVM.
Our main contribution is analysis of optimized bytecode in order to answer research question 2. In doing so, we addressed many structural differences in the bytecode-section, as we have outlined in the previous chapter, but also general bytecode-patterns, which the bytecode-optimizer modifies correspondingly.
The last chapter addresses the research-question 3 and describes a conceptual implementation

of a re-optimizer of EVM-bytecode. In order to demonstrate redundancy in the deployed bytecodes on the Ethereum blockchain, a re-optimizer is supposed to optimize deployed bytecodes to identify redundancy. This chapter provides a design for such an implementation as well as conceptual and technical challenges and drawbacks of such an approach.

7.2 Future Work

One main goal for the future could be the implementation of the actual re-optimizer. The upcoming challenges mentioned above would be needed to be overcome or, as a trade-off, the selection of bytecodes to be optimized would have to be narrowed down. For example, bytecodes with special assembly-instructions may eventually need to be ignored. The trade-off between the complexity of the implementation of the parser and the processing of the bytecodes into assembly-items is a trade-off that requires different considerations.
With this re-optimizer all deployment bytecodes on the Ethereum blockchain could be optimized and the resulting re-optimized bytecodes could be scanned for redundancy on the Ethereum blockchain. Such a search for redundancy could be used to help many studies that work with their own data-sets of unique smart-contract bytecodes and don't take redundancy regarding missing optimization into consideration. Therefore narrowing down the sets of unique smart-contracts could lead to more accurate empirical results in the field of bytecode analytics of Ethereum.
Apart from this, it would be also possible to elaborate even further on the Yul optimizer, which is currently still in the development-phase and seems to play a more important role in the upcoming migration to web-assembly. Most notably, the various optimizers in the Yul optimizer are much more advanced and sophisticated. In this context it would be interesting to research how the order of the optimizers influences the optimization and how the order of the optimizers could be improved.

List of Figures

Bibliography

[1] G. Antonopoulos Andreas Wood. *Mastering Ethereum, Building Smart Contracts and DApps*. O'Reilly Media, Incorporated, 2018.

[2] G. Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* (2014).

[3] Y. Hirai. "Defining the Ethereum Virtual Machine for Interactive Theorem Provers". In: *International Conference on Financial Cryptography and Data Security* (2017).

[4] H. S. Galal and A. M. Youssef. "Trustee: Full Privacy Preserving Vickrey Auction on top of Ethereum". In: *Lecture Notes in Computer Science* (2019). ISSN: 1611-3349.

[5] S. Kindler. "Towards a Toolchain for Exploiting Smart Contracts on the Ethereum Blockchain". MA thesis. Technische Hochschule Ulm, 2019.

[6] K. GeunWoo. "Debugging RLP on the Web". In: *Medium.com* (2019).

[7] J. Krupp. "Teether: Gnawing at Ethereum to Automatically Exploit Smart Contracts". In: *The Advanced Computing System Association* (2018).

[8] W. Maximilian. "Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity". In: *University of Vienna* (2018).

[9] *Solidity Documentation (v0.7.4)*.

[10] M. Suiche. "Porosity: A Decompiler For Blockchain-BasedSmart Contracts Bytecode". In: *Comae Technologies* (2017).

[11] M. Zoltu. "Ethereum Improvement Proposal 161". In: *github.com* (2017).

[12] EthereumStackexchange. "Do contracts also have a nonce?" In: *ethereum.stackexchange.com* (2016).

[13] B. Vitalik. "Ethereum Whitepaper". In: (2013).

[14] D. Perez. "Broken Metre: Attacking Resource Metering in EVM". In: *Imperial College London* (2020).

[15] J. Davidson. "Simplifying code generation through peephole optimization". In: *The University of Arizona Libraries* (1981).

[16] A. Santander. "Deconstructing a Solidity Contract". In: *OpenZeppelin blog* (2018).