TU WIEN Informatics

Optimization of Solidity Smart Contracts

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Tamara Brandstätter, BSc
Matrikelnummer 01638971

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dr.-Ing. Stefan Schulte

Wien, 26. Februar 2020

Tamara Brandstätter	Stefan Schulte

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Optimization of Solidity Smart Contracts

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering & Internet Computing

by

Tamara Brandstätter, BSc
Registration Number 01638971

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dr.-Ing. Stefan Schulte

Vienna, 26th February, 2020

Tamara Brandstätter	Stefan Schulte

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Tamara Brandstätter, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 26. Februar 2020

Tamara Brandstätter

Acknowledgements

First of all, I would like to thank my advisor Stefan Schulte for his great support and guidance in the past months. In addition, I would like to thank Michael Borkowski, for the idea of the thesis and the support in the beginning.

I also thank my family, especially my parents Michaela and Peter, for always believing in me and supporting me to achieve my goals. A very special thank you goes to my boyfriend Michael, for his continuous support, endless patience and helpful discussions about my topic.

Kurzfassung

Smart Contracts sind Computerprogramme, die auf einer Blockchain bereitgestellt werden und dann ausgeführt werden können. Abhängig von der Größe des erzeugten Bytecodes, müssen für das Deployment Gebühren in Form von "Gaskosten" bezahlt werden. Bei der Ausführung eines bereits bereitgestellten Contracts sind weitere Gaskosten zu bezahlen, abhängig von den benötigten Rechenschritten sowie dem benötigten Speicherplatz. Im Fall von Ethereum werden jeden Tag Millionen USD in Form von Gaskosten gezahlt.

Diese Gaskosten steigen, sofern ein Contract nicht optimal implementiert wurde, d. h. nicht notwendige Rechenschritte enthält oder zu viel Speicherplatz benötigt wird. Daher besteht ein großer Anreiz, diese Gaskosten zu senken, um Geld zu sparen.

Wir haben 19 Optimierungsstrategien in Form von Regeln aus dem Bereich Software Engineering identifiziert, die auf Solidity Smart Contracts angewendet werden können, um die erforderlichen Gaskosten zu senken. Um zu zeigen, wie diese Regeln zur Optimierung von Solidity Smart Contracts verwendet werden können, haben wir einen Prototypen entwickelt. Dieser Prototyp erkennt Regelverletzungen von 9 dieser Regeln, wobei wir darüber hinaus für 6 dieser Regeln eine automatische Optimierung implementiert haben. Wir haben mithilfe des Prototyps 3,018 Opensource Smart Contracts von *etherscan.io* analysiert. In diesem Testdatensatz wurden 471 Regelverstöße in 204 unterschiedlichen Dateien gefunden. Das heißt, ca. 6.8% der Contracts aus unserem Testdatensatz verstoßen gegen zumindest eine der Regeln.

Wir haben alle Contracts, welche automatisch vom Protoypen optimiert wurden, in einer Testumgebung deployed, um den Gasverbrauch zu vergleichen. Wir konnten im Durchschnitt 1,213 Gas für das Deployment einer optimierten Version eines Smart Contracts einsparen. Für einen Funktionsaufruf einer optimierten Funktion, konnten wir, im Vergleich zu einer unveränderten Funktion, im Durchschnitt 123 Gas einsparen.

Abstract

Smart contracts are computer programs that are deployed on the blockchain and can then be executed. Fees in form of gas have to be paid for deploying a smart contract on the blockchain, depending on the size of the byte code. Additional fees have to be paid, whenever a function of a smart contract is executed, depending on the needed computational steps and required storage space. On the Ethereum blockchain, millions of USD are paid in form of gas fees every single day.

Since gas cost can increase if a contract is not well-implemented, there is a lot of incentive to reduce these gas cost in order to save money.

We identified 19 optimization strategies from the field of software engineering, which can be applied to Solidity smart contracts in order to reduce the required gas cost. To show how these rules can be used to optimize smart contracts, we developed a prototype. The prototype detects rule violations from 9 of these rules and in addition automatically optimizes 6 of them. The prototype analyzed 3,018 verified open source smart contracts from *etherscan.io*. We found 471 rule violations in our test data set spread across 204 different contract files. That means, roundabout 6.8% of the data set violated at least one of the rules.

We deployed the automatically optimized contracts in a test environment before and after the optimization, to compare the gas usage. We were on average able to save 1,213 gas for deploying an optimized contract version compared to the initial one. For calling an optimized function once, we were able to save on average 123 gas, compared to the initial function call.

Contents

CHAPTER 1

Introduction

Satoshi Nakamoto introduced the basic technologies behind today's cryptocurrencies in 2008 [Nak08]. One year later, the first cryptocurrency, Bitcoin, was launched. The main technology behind Bitcoin is a public distributed append-only ledger which stores all transactions that are propagated using a peer-to-peer (P2P) network. This ledger is known as the blockchain, since the transactions are grouped in blocks and chained together using cryptographic hash pointers. Grouping the transactions and appending the blocks on the ledger is the task of the so-called miners in the network [Zoh15].

Since the original Bitcoin proposal, cryptocurrencies and blockchains have gained a lot of attention. As the popularity of these technologies increased, a lot of alternative currencies, also known as altcoins, and other concepts in the blockchain field emerged [XWS19, Zoh15]. On the website *coinmarketcap*[1] over 2,000 different coins are listed today. Some of these newly developed currencies are extensions to the existing Bitcoin protocol, like for example Litecoin [Lee11]. Litecoin was built on top of Bitcoin and provides some changes to the original Bitcoin code. Other currencies originated more or less separately from Bitcoin and are subject to their own ideas and concepts. An example for one of these currencies is Ripple [AKM+15], which is a distributed payments network based on blockchain technologies and is used for money transfer between banks.

One of the most popular developments since the introduction of Bitcoin is the Ethereum project [W+14], which uses a separate blockchain with its native currency, called Ether. Ethereum currently holds the second highest market capitalization after Bitcoin[2]. The smallest denomination of Ether (ETH) is Wei[3]. As a significant improvement compared to Bitcoin, Ethereum introduced the possibility to write Turing-complete computer programs that can be executed by the nodes participating in the Ethereum network [Dan17].

[1] `https://coinmarketcap.com/all/views/all/`, accessed 22.07.2019
[2] `https://coinmarketcap.com/`, accessed 27.06.2019
[3] 1 Ether = 10^{18} Wei

These computer programs are called smart contracts and are written in a Turing-complete high-level programming language, like Solidity[4] or Vyper[5]. The high-level code is translated into bytecode and can then be deployed on the blockchain. Once deployed on the Ethereum blockchain, it is accessible to all nodes participating in the P2P network. Simply put, smart contracts are programs that are written in a high-level programming language, stored on the blockchain, accessible to the network participants and executable. The execution environment of Ethereum is the Ethereum Virtual Machine (EVM), which is a state machine with a stack-based architecture. Turing-complete computer programs bring some drawbacks for the executing node, like infinite loops. To avoid these downsides of Turing-completeness, Ethereum introduced a mechanism to set an upper limit for the computational steps that can be performed [W+14]. This limit is set on transaction level, for each transaction individually by the sender of the transaction. That means that every transaction executed in the EVM is guaranteed to terminate at a certain point in time, at the latest when the upper limit is reached. Therefore, the EVM is a quasi Turing-complete machine [W+14], since it executes Turing-complete programs, but has the additionally introduced limitation of computational steps.

The unit for defining the upper limit for a transaction is called gas. The gas is then used as "fuel" for the transaction. Every operation inside the EVM has a defined amount of gas it consumes during its execution. For example the *ADD* operation, which adds two numbers, consumes three amounts of gas. The highest gas consumption of 32,000 amounts has the operation *CREATE* which creates a new contract. The limit for a transaction, defined in units of gas, is set by the sender of the transaction. The amount of gas that was necessary to execute the transaction then has to be paid by the sender in form of transaction fees.

Gas exists only inside the virtual machine, an Ethereum account can not "own" gas. When gas has to be paid, it is converted and charged in form of Wei/Ether. To emit how much Ether one amount of gas is worth, the sender additionally has to define the gas price for the transaction. The gas price is the amount of Wei the sender is willing to pay for each amount of gas when it comes to fee charges.

When the transaction gets included in a block by a miner in the network, the amount of gas that was needed is emitted during the execution. With that help, the fee for the transaction can be calculated as a multiplication of the predefined gas price with the gas consumed. After an successful execution, the sender pays the fee in Ether to the executing miner. When a transaction runs out of gas, the transaction fails and the state changes get reverted. Since the whole provided gas was consumed during the execution, the maximum fee has to be paid to the miner, which is the whole provided gas times the gas price.

[4] `https://github.com/ethereum/solidity`, accessed 28.06.2019
[5] `https://github.com/ethereum/vyper`, accessed 28.06.2019

To get a feeling for the data that is actually on the blockchain, we give a real example and calculate the gas consumption from the Ethereum blockchain on 16.05.2019:

- Gas used[6]: 48.03 billion amounts
- Gas price[7] (on average): 21,005,575,115 Wei
- Ether price[8] (on average): 248.77 USD/ETH
- Ether spent on gas fees: 1,008,894,83 ETH
- USD spent on gas fees: 250,982,767.40 USD

That gives us over 250 millions of USD that are paid to miners in form of fees on a single day. When it comes to a high number of transactions, the gas price plays an important role in the Ethereum network. For example, on the 8th of December in 2016, the all-time high average gas price, at the time of writing, was reached, which was 939,588,332,579 Wei for one amount of gas, which is over 44 times higher than the gas price from the example day above. Since a significant part of these cost are controlled by contract code on the blockchain, we see high potential to reduce these cost with the help of code optimizations.

Program optimizations, or code optimizations, are modifications to the source code that are made to improve the code in order to reach a certain goal. For example, object-oriented design patterns, as presented by Gamma et al. [Gam95], can be a basis for code optimizations. So, when applying one of these design patterns to already existing code, the optimization process is improving the code in order to reach the goal of code re-usability and flexibility. In case of Solidity smart contracts, the object-oriented design patterns would also be applicable since it is an object-oriented programming language. So, re-usability and flexibility for Solidity smart contracts could be reached by applying object-oriented design patterns.

Another way to optimize code is to modify the source code in order to improve the efficiency. We understand that a program is more efficient when it needs less space or less time. Optimization approaches regarding efficiency are covered, e.g., by Bentley [Ben82]. When applying one of the proposed rules, the optimization process is improving the code in order to reduce the required space or reduce the time needed for execution. We can separate optimizations regarding efficiency into two sub fields: (a) improving the efficiency of algorithms that are used and (b) improving the efficiency with the help of the underlying system.

Work on analysing algorithms and how to make them efficient has already been done in the early 1970's [AH74]. This existing knowledge can be used to improve algorithms that are written in smart contract code.

[6] `https://etherscan.io/chart/gasused/`, accessed 28.06.2019
[7] `https://etherscan.io/chart/gasprice/`, accessed 28.06.2019
[8] `https://etherscan.io/chart/etherprice/`, accessed 28.06.2019

Understanding what happens underneath can also help to make further improvements regarding space and time. Therefore, in case of Solidity smart contracts, we have to investigate Ethereum's execution environment to understand how the different operations impact the required execution time or storage space.

1.1 Problem Description

The aim of this thesis is to make optimizations on existing smart contracts with the overall goal to reduce gas cost. Since Ethereum is today the most-widely used blockchain supporting smart contracts, this thesis covers Ethereum smart contracts. This is highly coupled with optimizations regarding efficiency, since gas cost increase with the required storage space and computation resources.

Since gas cost can increase significantly if a contract is not well-implemented, there is a lot of incentive to reduce these gas cost in order to save money. Therefore, in this thesis, we optimize with regard to efficiency. We give no attention to optimizations regarding reusable or flexible code.

There are different factors that influence the gas cost:

- **Computational steps:** Every computational step costs some defined amount of gas. Reducing the computational steps results in lower gas cost.
- **Storage space:** Expanding the storage costs some defined amount of gas. We differ between short-term and long-term storage. Reducing the required storage space and moving from long-term to short-term whenever possible will result in lower gas cost.
- **Byte code:** The fee for deploying a smart contract depends on the length of the generated byte code. Reducing the length of the byte code results in lower gas cost.

There have already been some approaches on reducing gas cost that will be described in Section 3. To the best of our knowledge, none of these existing approaches take optimization approaches from the field of software engineering into account. The focus lies only on optimizing with the help of understanding the underlying system. Therefore, within this thesis, we will investigate the impact of general optimization approaches and develop a strategy in form of guidelines in order to decrease the required gas cost. Our goal is that these guidelines can be applied generally when developing smart contracts in Solidity. A prototype for optimizing Solidity smart contracts is developed during the analysis. During the prototype implementation we will focus on high-level optimizations in order to develop a tool that supports the developer.

1.2 Thesis Structure

The thesis is structured as follows:

- **Chapter 2** provides the necessary background for the following chapters.
- **Chapter 3** gives an overview of the state of the art in optimizing Ethereum smart contracts.
- **Chapter 4** discusses how existing general optimization approaches from the field of software engineering can be applied to Solidity smart contracts and how these approaches affect the gas consumption. A detailed list of applicable optimization strategies will be given.
- **Chapter 5** gives an overview of the system design of our prototype and provides insights into the implementation.
- **Chapter 6** covers the evaluation of the implemented optimization strategies.
- **Chapter 7** concludes the work and discusses some insights and further improvements for the future work.

CHAPTER 2

Background

In this chapter, we explain the necessary background and used technologies for this thesis. We start with the definition of a blockchain and the main characteristics of the Ethereum blockchain. Afterwards, we discuss smart contracts and how they are executed in the execution environment of Ethereum, i.e., the EVM. Then, we explain the main parts of the execution environment of Ethereum with a focus on the gas mechanism. Readers already familiar with the Ethereum mechanics may skip this chapter.

2.1 Blockchain

A blockchain is a distributed append-only ledger which is used to store all transactions of the distributed P2P network. These transactions are grouped in blocks and chained together with the help of cryptographic hash pointers. The structure of a blockchain is shown in Figure 2.1. The hash pointers point to the previous data in the chain together with a hash value of the referenced data. When the data changes, the hash value of that data results in a different hash value[1]. With this help, it can be easily recognized if the data has been manipulated inside a block [NBF+16]. The transactions inside a block are organized in a transaction tree. The underlying data structure of this transaction tree is a modified Merkle Patricia Tree [W+14].

2.1.1 Ethereum Network

The Ethereum network is a distributed P2P network. There is no hierarchy between the nodes participating in the network. New transactions are broadcast through the network and recognized as pending transactions until they are included in an accepted block. We

[1] No hash function is collision free, but it is infeasible to find a collision. This fact can therefore be neglected.

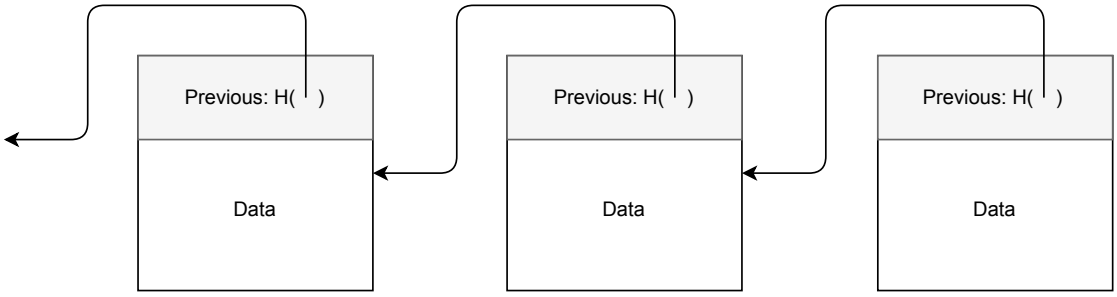


Figure 2.1: Structure of a Blockchain (source: [NBF+16])

differ between two node types that can participate in the network (a) light nodes and (b) full nodes [B+14].

Light Nodes

Light nodes, also known as light clients, are light-weight clients that do not validate the whole blockchain. Instead, they only validate those parts relevant for a client's transaction. Wallet software is an example of a light client. Wallets are used to store the public and/or private keys and interact with the blockchain.

Full Nodes

Full nodes, unlike light nodes, validate the whole blockchain. Therefore, they have to download the whole blockchain. Most miners run a full node. A miner has the goal to create new blocks that are appended to the ledger. Therefore, a miner collects all valid pending transactions that have been broadcast in the network before. In order to decide whether a transaction is valid or not, the miner has to know the whole blockchain. Then, the miner proposes a new block containing transactions of its choice and appends this block to the longest valid chain. The miner has high incentive to mine a valid block, since mining a valid block is rewarded with 2 ETH at the time of writing. Therefore, every miner should run a full node, even though it is not mandatory. Otherwise, there is no guarantee that the mined block is valid.

2.1.2 Accounts

Due to the existence of smart contracts on the Ethereum blockchain, we have to differ between two account types (a) externally-owned accounts and (b) contract accounts [W+14].

Externally-Owned Accounts

Externally-owned accounts have the same role as accounts in Bitcoin. They are controlled by their private key. When sending Ether to an address of an externally-owned account, the balance of this account gets increased.

Contract Accounts

Contract accounts, also known as internal accounts, represent accounts for smart contracts. They are controlled by their deployed code on the Ethereum blockchain [B+14]. When sending Ether to an address of a contract account, the code responsible for receiving Ether gets executed. Not all contracts can receive Ether by simply sending some Ether to the account address. That happens, when no function was implemented that handles receiving Ether.

2.1.3 Transactions

Every interaction of a client with the blockchain results in a transaction. This transaction gets broadcast through the P2P network and will then be validated and eventually added to the blockchain. In Ethereum, we differentiate between two types (a) transactions and (b) message calls [B+14].

Transaction

Transactions are signed data packages that are sent from an externally-owned account [B+14].

A transaction contains the following fields:

- The sender of the message
- The receiver of the message
- The value transferred from the sender to the receiver
- An optional data field
- STARTGAS
- GASPRICE

The sender has to sign the transaction, such that the sender can be identified through the signature. The value is the amount of Ether that is sent from the sender to the receiver, which can also be set to zero. The data field is optional and can be accessed by the execution environment. The STARTGAS field is set by the sender and represents the limitation of computational steps and storage space. It is the amount of gas the sender wants to use as an upper limit to perform the transaction. The GASPRICE is also set by the sender of the transaction and defines the price in Wei the sender is willing to pay for each amount of gas needed to perform the transaction.

Message Call

Message calls, also simply called messages, are sent from a contract account to another and therefore only exist within the execution environment [B+14]. A message always has a transaction as its origin.

A message contains the following fields:

- The sender of the message
- The receiver of the message
- The value transferred from the sender to the receiver
- STARTGAS

The sender is implicitly set in a message call. For the two fields, value and STARTGAS, the same rules apply as for transactions. The field GASPRICE is missing, since the price is already defined by the origin transaction.

2.2 Smart Contracts

In 1997, Szabo [Sza97] and Miller [Mil97] presented the idea of smart contracts while making assumptions about the future law system. They had the idea to formalize contract agreements as a digital set of promises and protocols. These agreements could then be enforced with the help of computer algorithms. Smart contracts nowadays refer to computer programs that are deployed on a blockchain. Once deployed on the blockchain, these programs are available for all network participants.

2.2.1 Ethereum Smart Contracts

Ethereum smart contracts are written in a high-level programming language and translated into EVM bytecode. The most commonly used high-level programming language is called Solidity[2]. The generated EVM bytecode can then be executed by the EVM. When the code is deployed on the Ethereum blockchain, no more changes can be made to that code. Hence, smart contracts deployed on the blockchain are often referred to autonomous agents that live inside the Ethereum execution environment [B+14].

Token Contracts

Token contracts are special smart contracts that are developed in a way to represent another currency or trade-able good inside the Ethereum blockchain. These contracts make it possible for anyone to set another layer of currency on top of the existing Ethereum blockchain.

[2] `https://github.com/ethereum/solidity`, accessed 06.07.2019

2.3 Ethereum Virtual Machine

The EVM is the execution environment of Ethereum. It is a quasi-Turing complete state machine with a stack-based architecture. The quasi-Turing completeness comes from the limitation of computational steps that is set by the initiator of a transaction. This limitation has been introduced to overcome potential issues like infinite loops [W+14]. Without this limitation, nodes could get stuck when executing Turing-complete programs that may not terminate. Written smart contracts are translated into bytecode that can be executed in the EVM. This code is executed on the physical hardware of every validating node in the network that validates a transaction that makes use of the deployed contract code [B+14].

2.3.1 State Machine

Ethereum is a transaction-based state machine. Starting in the initial state and applying all transactions that happened until now, gives us the current state. This state transition function Υ is defined as

$$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T) \tag{2.1}$$

where σ_{t+1} is the new world state when applying the transaction T on the current world state σ_t [W+14]. σ represents the world state, also known as global state, which is a mapping between addresses and account states $a \rightarrow \sigma[a]$. The account state $\sigma[a]$ of any address a stores the following values:

- **nonce** $\sigma[a]_{\mathbf{n}}$: If a is an externally-owned account, the nonce is the number of transactions initiated from this address. If a is a contract account, the nonce is the number of contract creations made by this account.
- **balance** $\sigma[a]_{\mathbf{b}}$: The balance of the account in Wei.
- **storageRoot** $\sigma[a]_{\mathbf{s}}$: The hash of the storage root node.
- **codeHash** $\sigma[a]_{\mathbf{c}}$: If a is an externally-owned account, the codeHash is the Keccak-256 hash of the empty string. If a is a contract account, the codeHash is the hash of the account's EVM code.

2.3.2 Storage Types

The EVM has access to three different storage types [B+14]:

- **Stack**, which is used for executing the computation. The values on the stack are reset after the computation has finished.
- **Memory**, which is an infinitely expandable word-addressable word array. The word size in the EVM is 256 bits. The memory is also volatile and therefore resets after the computation has finished.

- **Storage**, which is the long-term persistent storage type and therefore the most expensive. It is a key-value store and part of the world state.

2.3.3 Gas Mechanism

The limitation of computational steps during the execution of a transaction is reached with the help of the introduced gas mechanism. Each EVM operation has a defined gas usage. An excerpt of some EVM operations is given in Table 2.1, a complete overview can be found in [W+14]. There exist two EVM operations that allow to refund gas, which are listed in Table 2.2. Within a transaction, it is not possible to obtain Wei with the help of refund operations. In the best case, no gas has to be paid for that transaction.

The limitation is set by the sender of a transaction, by defining the two values: STARTGAS and GASPRICE. STARTGAS is the amount of gas defined as an upper limit the transaction is allowed to consume. GASPRICE sets the price in Wei that the sender has to pay for each amount of consumed gas. The amount of gas given to a transaction is also called the fuel of this transaction. The GASUSED is the used gas and is defined after finishing the execution of the transaction. The GASUSED multiplied by the GASPRICE gives the gas fee and has to be paid to the miner.

There are three possible outcomes that can happen when executing a transaction:

- **STARTGAS = GASUSED**: The defined upper limit is the exact amount of gas that was needed for performing the transaction. Therefore, the miner who publishes the block containing this transaction gets the maximum fee of STARTGAS * GASPRICE.
- **STARTGAS > GASUSED**: The defined upper limit of gas was higher than the needed amount of gas. Therefore, the miner who publishes the block containing this transaction gets the fee of GASUSED * GASPRICE. The remaining gas fees (STARTGAS - GASUSED) * GASPRICE get paid back to the sender of the transaction.
- **STARTGAS < NEEDEDGAS**: The gas that would be needed to perform the whole transaction exceeded the provided STARTGAS. Therefore, the transaction fails and the miner who publishes the block containing this transaction gets the maximum fee of STARTGAS * GASPRICE. When failing, the last computation gets reverted.

Table 2.1: EVM operation gas cost overview, from: [W+14]

Name	Value	Description
G_{zero}	0	Paid for STOP, RETURN or REVERT operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
$G_{verylow}$	3	Paid for ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH*, DUP* or SWAP* operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
G_{low}	5	Paid for MUL, DIV, SDIV, MOD, SMOD or SIGNEXTEND operation.
G_{mid}	8	Paid for ADDMOD, MULMOD or JUMP operation.
$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
G_{sload}	200	Paid for a SLOAD operation.
G_{call}	700	Paid for a CALL operation.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
$G_{transaction}$	21000	Paid for every transaction.
G_{create}	32000	Paid for a CREATE operation.
$G_{txcreate}$	32000	Paid by all contract-creating transactions.

Table 2.2: EVM operation gas refund overview, from: [W+14]

Name	Value	Description
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.

CHAPTER 3

State of the Art

Some research in the field of optimizing smart contracts have already been done so far. We will separate the state of the art in smart contract optimizations regarding gas cost into (a) high-level optimizations, (b) low-level optimizations and (c) general optimizations that were not designed specifically for smart contracts.

3.1 High-level Smart Contract Optimizations

High-level optimizations depend on the used high-level programming language and therefore aim to optimize code at the time it is written. To the best of our knowledge, the research regarding smart contract optimizations done so far only considers the high-level programming language Solidity.

3.1.1 Chen et al., Under-optimized Smart Contracts Devour Your Money

Chen et al. [CLLZ17] conducted the first optimization-related investigation on Solidity smart contracts. The work describes possible gas cost reductions on a high level with a focus on transaction cost. They identify seven gas-costly anti-patterns that consider reductions for transactions and reductions for deploying new smart contracts on the blockchain. The patterns are divided into two categories: (a) useless-code-related patterns and (b) loop-related patterns. We give an overview of the identified anti-patterns and the solution how to get rid of them in Table 3.1.

They propose a tool named "GASPER" that identifies three of these anti-patterns in Solidity code. GASPER is based on the tool OYENTE [LCO+16] for Control-flow graph (CFG) generation. The tool has so far not been made public in any form [dAS19].

Hence, we can not use the implementation of GASPER for our prototype, but we can use the described anti-patterns as a basis for our analysis and build on top of them.

Table 3.1: High-level anti-patterns in Solidity smart contracts

Category	Problem	Solution
Useless code	Conditions that evaluate to true under all circumstances.	Remove the condition.
Useless code	Conditions that evaluate to false under all circumstances.	Remove the condition and the body, since it will never be reached.
Loop	Expensive loop operations when storage variables are accessed within a loop.	Move the access of the variable outside the loop.
Loop	Constant outcome of a loop.	Remove the whole loop and return the result immediately.
Loop	Several loops that could be combined.	Loop fusion which means combining several loops to one.
Loop	Repeated computations in a loop.	Compute once and reuse the result in the loop.
Loop	Conditions in a loop that always have the same outcome.	Move the conditions to a place before the loop.

3.1.2 Signer, Gas Cost Analysis for Ethereum Smart Contracts

Signer [Sig18] presents a tool named "VisualGas" that can be used to visualize gas cost in depth for supporting the developer with an overview how the found cost have their roots in the code. The focus here lies on the visualization of used gas in order to support the developer. No automated optimizations or suggestions how to optimize the code are made. VisualGas makes some impact for high-level optimizations, since it is a tool to identify gas-costly parts in the code.

3.1.3 Hukkinen, Reducing Blockchain Transaction Costs in a Distributed Energy Market Application

Hukkinen [Huk18] focuses on optimizing a specific contract of the energy market in his work. He has developed five guidelines for optimizing gas consumption of an Ethereum smart contract. These guidelines describe on a high level which actions can lead to high gas costs and should be avoided whenever meaningful:

1. **Contract creation:** Creating a contract costs 32,000 amounts of gas. Therefore, it is better to encapsulate as much functionality as useful in a single contract instead of deploying, for instance, a single contract for every user.

2. **Transaction:** Creating a transaction costs 21,000 amounts of gas. Therefore, it is better to use fewer functions with more actions instead of small functions.

3. **Store a word:** Storing a word in the persistent storage costs 20,000 amounts of gas. Therefore, the solution is to use memory instead of persistent storage whenever possible or to store data even off-chain.

4. **Delete contract:** Deleting a contract refunds 24,000 amounts of gas. Therefore, unused contracts should be deleted whenever meaningful.

5. **Release storage:** Releasing storage refunds 15,000 amounts of gas. Therefore, unused storage space should be released whenever meaningful.

6. **Off-chain solutions:** A general approach that can be followed in order to save gas is to use off-chain transactions whenever possible.

This work focuses on migrating from on-chain execution to off-chain execution which will not be an option in our case, since we will not consider off-chain solutions.

Hukkinen applied his developed guidelines on a smart contract of the energy market. As a result of the work, the total used gas fees for this certain use case were reduced by 11%. No additional tool was implemented during the research.

We will take into account the guidelines that do not rely on migrating on-chain to off-chain solutions. Since the guidelines were developed for a manual application, we can not use them for our automated optimization prototype, but we can use them again as a basis of our analysis.

3.2 Low-level Smart Contract Optimizations

Low-level optimizations aim to optimize the generated bytecode of smart contracts. Therefore, the optimization techniques are independent from the used high-level programming language.

3.2.1 Chen et al., Towards Saving Money in Using Smart Contracts

Chen et al. [CLZ+18] proposed a follow-up paper of [CLLZ17] where they focus on low-level optimizations. They developed 24 anti-patterns in the EVM bytecode for contract deployment and transactions. The five presented anti-patterns in the paper are:

1. Two consecutive swap(X)s can be deleted since the execution has no effect.

$$\{swap(X), swap(X)\} \rightarrow delete$$

   Six amounts of gas can be saved for each of these patterns, since a swap operation ($G_{verylow}$) consumes three amounts of gas, as it can be seen in Table 2.1.

2. M consecutive jumpdest(X)s can be deleted since the target will always be the destination of the final jumpdest.

$$\{Mconsecutivejumpdest\} \rightarrow jumpdest$$

The execution of a jumpdest operation ($G_{jumpdest}$) requires 1 amount of gas, as it can be seen in Table 2.1, so a saving of M-1 gas amounts is possible.

3. Pop after an Operation can be reduced to pop.

$$\{OP, pop\} \rightarrow \{pop\}$$

$$OP \in \{iszero, not, balance, calldataload, extcodesize, blockhash, mload, sload\}$$

OP consumes the top stack element, executes the operation and pushes the result to the stack. When followed by a pop operation, it is semantically identical to just performing a pop. The gas saving depends on the operation.

4. One swap can be removed for a commutative operation.

$$\{swap1, swap(X), OP, dup(X), OP\} \rightarrow \{dup2, swap(X+1), OP, OP\}$$

$$2 <= X <= 15, OP \in \{add, mul, and, or, xor\}$$

A saving of > 3 amounts of gas can be achieved.

5. OP before stop can be reduced to stop.

$$\{OP, stop\} \rightarrow \{stop\}$$

OP can be any operation excluding jump operations and operations that make changes in the storage. The gas saving depends on the operation.

Chen et al. also propose a tool named "GasReducer" that automatically detects anti-patterns in Ethereum bytecode and replaces them via bytecode-to-bytecode optimization. Chen et al. investigated all deployed smart contracts on the Ethereum blockchain as of June 2017 and processed them with the help of the tool GasReducer. The investigation has revealed that 2,040,892,224 units of gas are wasted for deployment and 7,185,048,532 units of gas are wasted for invocations of smart contracts.

The tool is not open source, so we can not use it for our further analysis, but the authors provided a list[1] of all identified anti-patterns. Even though the main focus of our prototype will be on high-level analysis, we can use these low-level anti-patterns and apply them on a high-level basis.

[1] `https://goo.gl/wJpAZ6` accessed 25.04.2019

3.2.2 Albert et al., EthIR: A Framework for High-Level Analysis of Ethereum Bytecode

Albert et al. [AGL+18] present a framework named EthIR[2] for high-level analysis of Ethereum bytecode, that generates a control flow graph from the bytecode. This relies on the tool OYENTE[3] – a symbolic execution tool for security bugs in smart contracts, proposed by Luu et al. [LCO+16]. This tool can be used to gain deeper knowledge of the Ethereum bytecode on a high-level basis. We won't use it for our prototype implementation, since we focus on the source code on a high level instead of the byte code.

3.3 General Optimizations

Optimization approaches in the area of software engineering that were not developed with a focus on smart contracts will be further called general optimizations.

3.3.1 Bentley, Writing Efficient Programs

Bentley [Ben82] developed a set of rules to optimize programs that could be useful for optimizing Ethereum smart contracts. The rules are separated into six categories:

1. **Space-for-Time-Rules:** Optimization is achieved by increasing the required space in order to decrease the runtime of a program.
2. **Time-for-Space-Rules:** Optimization is achieved by decreasing the required space and instead increasing the execution time because of necessary re-computation of information.
3. **Loop Rules:** Optimization is achieved by changing or removing loops.
4. **Logic Rules:** These rules deal with logic evaluations that test the program state.
5. **Procedure Rules:** These rules deal with the underlying structure of a program that is organized in procedures.
6. **Expression Rules:** These rules deal with expression optimizations like reusing results or replacing expensive expressions with cheaper ones.

Some of the loop rules described by Bentley are also identified by Chen et al. [CLLZ17], but not referenced as one of the general rules from Bentley. Therefore we think that in general optimizations there is high potential that it can be reused. To the best of our knowledge, none of the existing optimization work for smart contracts deliberately takes general optimizations into account. Therefore, we will investigate the rules described above, if and how they can be used to optimize Solidity smart contracts.

[2] https://github.com/costa-group/ethIR accessed 25.04.2019
[3] https://github.com/melonproject/oyente accessed 25.04.2019

Table 3.2: Categorization of existing approaches for gas cost optimizations

Author, title of the paper	Low	High	Deploy	Develop
Chen et al., Towards saving money in using smart contracts [CLZ+18]	x		x	x
Chen et al., Under-optimized smart contracts devour your money [CLLZ17]		x	x	x
Hukkinen, Reducing blockchain transaction costs in a distributed energy market application [Huk18]		x		x
Signer, Gas cost analysis for Ethereum smart contracts [Sig18]		x		x

3.4 Comparison

To the best of our knowledge, there is no gas cost analysis so far that combines both approaches – optimizing the code itself on a high-level basis in order to reduce the gas cost and then additionally analysing on a low-level the bytecode in order to get the best possible solution. A categorization of the gas cost analysing papers discussed above can be found in Table 3.2.

The column "Deploy" stands for optimizations regarding gas cost that affect the fees that have to be paid when deploying a new smart contract on a blockchain. Therefore, the gas cost have to be paid once during the deployment of the contract. A contract creation (G_{create}) costs 32,000 amounts of gas, as it can be seen in Table 2.1. Additionally, every byte of the contract bytecode costs 200 amounts of gas. Therefore optimizations regarding the length of the bytecode lead to smaller amounts of gas that have to be paid.

The column "Develop" stands for optimizations regarding gas cost that affect the fees for performing transactions that make use of the deployed smart contracts. These fees have to be paid periodically, for each interaction with the contract. A transaction call ($G_{transaction}$) costs 21,000 amounts of gas, as it can be seen in Table 2.1. Additionally, every operation and storage extension costs some additional amount of gas. Therefore, optimizations regarding (a) reductions of computational steps and (b) reductions of the used storage lead to smaller amounts of gas that have to be paid.

3.5 Conclusion

To the best of our knowledge, none of the existing smart contract optimization approaches take general optimization approaches into account. Therefore we will analyze and discuss how well existing general optimization approaches, as developed by Bentley [Ben82], can be applied on Solidity smart contracts.

Additionally, there exists no published tool so far that can support the developer on a high-level with automated Solidity code optimizations. Therefore, we will develop a

prototype for automated Solidity code optimization. As a basis for our prototype, we will analyze the list of anti-patterns as described by Chen et al. [CLLZ17, CLZ+18] and extend it with additionally applicable optimization rules. The result should be a list of optimization strategies with its potential gas savings. This list is then prototypically implemented as a proof-of-concept.

CHAPTER 4

Optimization Strategies

In this chapter, we discuss whether or not the optimization rules, as described by Bentley [Ben82], can be applied to Solidity smart contracts. We develop examples and examine how these rules affect the gas consumption. Examples taken from the book [Ben82] are explicitly marked. For the development of the examples and further analysis, we use the Remix IDE[1]. We compile our contracts with the latest Solidity compiler version 0.5.13 [sol] at the time of writing. Remix has a built-in JavaScript virtual machine (VM) on which we can deploy and run our examples. This also allows us to examine the gas cost of the performed transactions. The cost within a transaction in Solidity are separated into transaction cost and execution cost. Transaction cost are the total cost, including the cost of sending data to the blockchain. The execution cost are included in the transaction cost and represent the cost used for execution within the EVM. Referring to a response to a question on the Ethereum StackExchange[2] platform, the following operations are not part of the execution cost:

- $G_{txcreate}$: cost of a contract deployment (32,000 amount of gas)
- $G_{transaction}$: base cost of a transaction (21,000 amount of gas)
- $G_{txdatazero}$: cost for every zero byte of data (4 amount of gas)
- $G_{txdatanonzero}$: cost for every non-zero byte of data (68 amount of gas)

All remaining operations are included as fees in the execution cost. We will focus mainly on recurring cost, which means executing the corresponding functions within transactions. We will always start a new transaction to call a function and calculate the cost of the entire transaction, including the transaction overhead.

[1] `https://remix.ethereum.org`, accessed 09.10.2019
[2] `https://ethereum.stackexchange.com/a/29560`, accessed 12.10.2019

The rules are divided into six main categories and will be discussed below. Note that we do not break the calculation down to every single gas unit, but focus on the differences between the optimized and non-optimized contract code.

The Solidity compiler[3] has a built-in optimizer that can be activated by a flag. By default, this optimizer optimizes the contract for 200 calls over its lifetime. We will discuss the rules and develop examples without setting the optimization flag to true. For those rules that we identify as applicable for Solidity smart contracts, we also discuss whether the optimizer has an impact on the outcome or not.

For those optimizations, where the deployment cost are higher in the optimized version, we calculate after how many function calls the initial investment pays off. If we have to call the function at least 20 times to return the investment, we consider this optimization unsuitable for our prototype implementation.

4.1 Space-for-Time Rules

The optimizations in this category are achieved by increasing the space required to shorten the runtime of a program. The main approach is to store redundant information to reduce the runtime. As explained in Section 2, storage space in the EVM is expensive compared to other operations. Nevertheless, we will examine more deeply when these rules can be applied.

4.1.1 Space-for-Time Rule 1: Data Structure Augmentation

The time is reduced by expanding the structure with additional information or by changing the information so that it is easily accessible. One way to apply this rule to Solidity smart contracts are structs. In Solidity, structs are used to define new types [sol]. An example contract for applying this rule might look like this:

Listing 4.1: Code example space-for-time rule 1: Data structure augmentation

```
contract SpaceForTimeRule1 {
    struct TestStruct {
        uint256 size;
    }

    TestStruct s;

    function getSizeTimes2() public returns(uint256) {
        s = TestStruct(1234);
        return s.size * s.size;
    }
}
```

[3] https://solidity.readthedocs.io/en/v0.5.12/using-the-compiler.html, accessed 10.11.2019

```
contract SpaceForTimeRule1_opt {
    struct TestStruct {
        uint256 size;
        uint256 size2;
    }

    TestStruct s;

    function getSizeTimes2() public returns(uint256) {
        s = TestStruct(1234, 1234 * 1234);
        return s.size2;
    }
}
```

In the first contract, we introduce a struct named *TestStruct*, which contains an integer value. To execute the function *getSizeTimes*2(), we need to access the struct field and do the multiplication. In the optimized contract, we apply the space-for-time rule 1 and therefore add an additional field to the struct. To execute the function *getSizeTimes*2(), we no longer need to execute the multiplication and instead access the additional field to get the answer.

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions through transactions are listed in Table 4.1.

Table 4.1: Gas cost estimate space-for-time rule 1: Data structure augmentation

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	101,131	35,087	94,681	30,081
Deployment contract 2	104,947	37,887	97,635	32,287
Difference	**3,816**	**2,800**	**2,954**	**2,206**
Function call contract 1	41,952	20,680	41,496	20,224
Function call contract 2	61,777	40,505	61,526	40,254
Difference	**19,825**	**19,825**	**20,030**	**20,030**

The difference in the deployment cost of contract 1 and contract 2 is due to the additional non-zero bytes introduced in contract 2. The value we are interested in, is the function call. In our optimized version, we needed over 20,000 more gas than for the non-optimized version. This is because an additional store operation was required for the second field, which is worth 20,000 amounts of gas. Even if we moved the initialization of *TestStruct* inside the constructor, this would only shift the cost of the function call to the deployment cost. This reduces recurring cost, but is much higher for deployment than without the additional field.

Whenever the data type extension saves more gas than needed to store an extra field, this rule may apply. However, since most operations are much cheaper compared to storage expansion, which costs 20,000 gas amounts, we conclude that rule 1 generally is not applicable for Solidity smart contracts when used with storage. If we run the same example with the Solidity compiler optimizer enabled, the total gas cost are lower, but the ratio is almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer, which we expected because we identified this rule as inapplicable.

4.1.2 Space-for-Time Rule 2: Store Precomputed Results

The time is reduced by storing the results of an expensive computation to run them only once. All further requests return the result from a lookup table instead of recalculating the function. We will test the rule on a contract to calculate Fibonacci numbers, as mentioned in the book [Ben82] as an example.

The simple contracts can look like this:

Listing 4.2: Code example space-for-time rule 2: Store precomputed results

```
contract SpaceForTimeRule2 {
    function getSumOfFibonacci(uint256 digit) public returns(uint256) {
        uint256 sum = 0;
        for (uint256 i = 0; i < digit; i++) {
            sum += getFibonacci(i);
        }
        return sum;
    }

    function getFibonacci(uint256 n) private returns(uint256) {
        if (n == 0 || n == 1) { return n; }
        return getFibonacci(n - 1) + getFibonacci(n - 2);
    }
}

contract SpaceForTimeRule2_opt {
    uint256[] public precomputedNumbers;

    constructor() public {
        uint256[] memory lookupTable = new uint256[](10);
        for (uint256 i = 0; i <= 10; i++) {
            lookupTable[i] = getFibonacci(i);
        }
        precomputedNumbers = lookupTable;
    }
```

```
    function getSumOfFibonacci(uint256 digit) public returns(uint256) {
        uint256 sum = 0;
        for (uint256 i = 0; i < digit; i++) {
            sum += precomputedNumbers[i];
        }
        return sum;
    }

    function getFibonacci(uint256 n) private returns(uint256) {
        if (n == 0 || n == 1) { return n; }
        return getFibonacci(n - 1) + getFibonacci(n - 2);
    }
}
```

In the first contract, we introduce the function *getSumOfFibonacci()*, which takes a number as an argument and calculates the corresponding Fibonacci number. No matter how many times this function is called, the Fibonacci number is recalculated with each call. In the optimized version of the contract, we introduce an additional field *precomputedNumbers*, which represents the lookup table of the Fibonacci numbers. In the constructor, we calculate the first 10 Fibonacci numbers and store them in our lookup table.

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions with the parameter 9 through transactions are listed in Table 4.2.

Table 4.2: Gas cost estimate space-for-time rule 2: Store precomputed results

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	126,413	53,905	116,551	46,499
Deployment contract 2	404,702	306,330	381,513	289,941
Difference	**278,289**	**252,425**	**264,962**	**243,442**
Function call contract 1	40,719	19,255	39,463	17,999
Function call contract 2	26,745	5,281	26,647	5,183
Difference	**-13,974**	**-13,974**	**-12,816**	**-12,816**

The difference between the deployment cost of contract 1 and contract 2 results from the additional storage requirements of the second contract and the cost for pre-calculating the first 10 Fibonacci numbers. We needed about 14,000 fewer gas units to call the optimized contract function. The deployment cost for the second contract with the lookup table are over 250,000 gas units more expensive then the deployment cost for the first contract.

To calculate after how many function calls the initial investment pays off, we examine the cost when the optimizer is disabled and in the other case when the optimizer is enabled separately. When the optimizer is disabled, we have to invest 278,289 more gas for deploying the optimized contract and are able to save 13,974 gas for each function call. If we divide the additionally invested gas for the deployment by the gas saved for a function call, we come to the conclusion that the function must be called at least 20 times in order to achieve savings through this optimization. With the optimizer enabled, we have to invest 264,962 more gas for deploying the optimized contract and are able to save 12,816 gas for each function call. If we divide the additionally invested gas for the deployment by the gas saved for a function call, we come to the conclusion that the function must be called at least 21 times in order to achieve savings through this optimization.

Since storage operations are among the most expensive, this rule is not considered applicable for Solidity smart contracts. The pre-computation may save some gas in the future, but the initial effort is far too high and access to the precomputed results is still very expensive. If we run the same example with the Solidity compiler optimizer enabled, the total gas cost are lower, but the ratio is almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer, which we expected because we identified this rule as inapplicable.

4.1.3 Space-for-Time Rule 3: Caching

The idea behind the caching rule is that access to the most frequently accessed data is the cheapest. A way to implement caches in Solidity smart contracts is to load the most frequently accessed data from the persistent storage to the volatile memory. We give a simple example of how a cache can work in Solidity in Listing 4.3.

In the first contract, we introduce a storage variable *value* that contains a value that is set in the constructor. In the function *getValueFourTimes()*, we access this storage variable four times. In the second contract, we introduce the same storage variable *value*. In the function *getValueFourTimes()*, instead of accessing the storage variable four times, we introduce a caching variable in memory *val*. Therefore, we only access the storage variable once and instead access the memory variable four times.

Listing 4.3: Code example space-for-time rule 3: Caching storage values into memory

```
contract SpaceForTimeRule3 {
    uint256 value;

    constructor(uint256 val) public {
        value = val;
    }

    function getValueFourTimes() public returns(uint256) {
        return value + value + value + value;
    }
}
```

```
contract SpaceForTimeRule3_opt {
    uint256 value;

    constructor(uint256 val) public {
        value = val;
    }

    function getValueFourTimes() public returns(uint256) {
        uint256 val = value;
        return val + val + val + val;
    }
}
```

Estimated gas cost for deploying these contracts in the Remix JavaScript VM with the initial value 12345 for *expensive* and calling the corresponding functions through transactions are listed in Table 4.3.

Table 4.3: Gas cost estimate space-for-time rule 3: Caching storage values into memory

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	117,768	49,612	113,063	46,959
Deployment contract 2	117,896	49,612	113,063	46,959
Difference	**128**	**0**	**0**	**0**
Function call contract 1	22,281	1,009	21,666	394
Function call contract 2	21,694	422	21,666	394
Difference	**-587**	**-587**	**0**	**0**

The difference in the deployment cost of the contracts is due to the additional non-zero bytes introduced in the optimized version. The execution cost for both deployments are the same because the same operations are performed. The function call of the optimized version is much cheaper than that of the non-optimized version. Even if we imagine in the above example that the value is accessed only once, the function call in the optimized contract would cost only 13 units of gas more than the function call in the non-optimized version, since the introduced cache is very cheap.

When the optimizer is disabled, we have to invest 128 more gas for deploying the optimized contract and are able to save 587 gas for each function call. Therefore, the additional investment for the deployment of the contract pays off after the first function call.

Accessing the storage (SLOAD) costs 200 gas units, accessing the memory (MLOAD) costs only 3 gas units. Because storage access is far more expensive than accessing a memory variable, caching storage values that are accessed multiple times in a function

is always applicable to Solidity smart contracts. If we run the same example with the Solidity compiler optimizer enabled, the gas required to call the function of both contracts is exactly the same. Therefore, we derive that this optimization approach is already implemented in the optimizer.

4.1.4 Space-for-Time Rule 4: Lazy Evaluation

Lazy evaluation aims to avoid unnecessary evaluations and therefore postpones the evaluation to the latest possible point. This mechanism, also called call-by-need, was first introduced in the lambda calculus and is well established in the field of functional programming [Hug89]. We test the rule on a contract that calculates Fibonacci numbers, as mentioned in the book [Ben82]. Compared to the above example from the space-for-time rule 2, in which we calculated the top 10 Fibonacci numbers, this program calculates only the Fibonacci numbers that are actually needed and stores them in a lookup table for further requests.

Listing 4.4: Code example space-for-time rule 4: Lazy evaluation

```
contract SpaceForTimeRule4 {
    function getSumOfFibonacci(uint256 digit) public returns(uint256) {
        uint256 sum = 0;
        for (uint256 i = 0; i < digit; i++) {
            sum += getFibonacci(i);
        }
        return sum;
    }

    function getFibonacci(uint256 n) private returns(uint256) {
        if (n == 0 || n == 1) {
            return n;
        }
        return getFibonacci(n - 1) + getFibonacci(n - 2);
    }
}

contract SpaceForTimeRule4_opt {
    mapping(uint256 => uint256) public precomputedNumbers;

    function getSumOfFibonacci(uint256 digit) public returns(uint256) {
        uint256 sum = 0;
        for (uint256 i = 0; i < digit; i++) {
            sum += getFibonacci(i);
        }
        return sum;
    }
```

```
    function getFibonacci(uint256 n) private returns(uint256) {
        if (n == 0 || n == 1) {
            return n;
        }
        uint256 num = precomputedNumbers[n];
        if (num != 0) {
            return num;
        }
        uint256 fibonacciNumber = getFibonacci(n - 1) + getFibonacci(n - 2);
        precomputedNumbers[n] = fibonacciNumber;
        return fibonacciNumber;
    }
}
```

The first contract is the same as in the space-for-time rule 2 example above. In the optimized version of the contract, we introduce an additional mapping *precomputedNumbers*, which is filled and reused during function calls. When we request a Fibonacci number in the optimized contract version, we first check in our lookup table (*precomputedNumbers*), to see if the requested number has been previously calculated. In this case, the number of the lookup table is returned. Otherwise, the Fibonacci number is calculated and stored for future requests in the lookup table. In the optimized version, we only calculate the Fibonacci numbers that are actually needed for the computation. In the above example (space-for-time rule 2), we may have wasted computational power on the calculation of numbers we may never need.

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions with the parameter 9 through transactions are listed in Table 4.4.

The difference between the deployment cost of contract 1 and contract 2 results from the additional storage requirements of the second contract. When comparing the cost of the function call, we needed about 140,000 more amounts of gas to call our optimized function. This is due to the expensive operations of writing and reading from the storage. If we call the same function a second time, the first contract will still need 40,719 amounts of gas for the transaction. On the other hand, our optimized contract requires only 29,784 amounts of gas for the second function call (8,320 amounts of gas for execution). For a second request with the same values, we would then need about 10,000 fewer gas units.

To calculate after how many function calls the initial investment pays off, we examine the cost when the optimizer is disabled and in the other case when the optimizer is enabled separately. When the optimizer is disabled, we have to invest 48,820 more gas for deploying the optimized contract. We additionally have to invest 141,411 more gas for the first function call and are able to save 10,935 gas for each further function call. If we sum up the additionally invested gas for the deployment and the first function call and divide the sum by the gas saved for each additional function call, we come to the conclusion that the function must be called at least 19 times in order to achieve savings through this optimization. With the optimizer enabled, we have to invest 32,528 more gas for deploying the optimized contract. We additionally have to invest 141,874 more

Table 4.4: Gas cost estimate space-for-time rule 4: Lazy evaluation

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	126,413	53,905	116,487	46,499
Deployment contract 2	175,233	91,541	149,015	71,923
Difference	**48,820**	**37636**	**32,528**	**25,424**
1. Function call contract 1	40,719	19,255	39,463	17,999
1. Function call contract 2	182,130	160,666	181,337	159,873
Difference	**141,411**	**141,411**	**141,874**	**141,874**
2. Function call contract 1	40,719	19,255	39,463	17,999
2. Function call contract 2	29,784	8,320	29,401	7,937
Difference	**-10,935**	**-10,935**	**-10,062**	**-10,062**

gas for the first function call and are able to save 10,062 gas for each further function call. If we sum up the additionally invested gas for the deployment and the first function call and divide the sum by the gas saved for each additional function call, we come to the conclusion that the function must be called at least 19 times in order to achieve savings through this optimization.

This rule can always be applied if the number of accesses is large enough to justify the overhead of storing the precomputed values in a lookup table. This question must be answered separately for each case. However, we conclude that the space-for-time rule 4 is generally applicable for Solidity smart contracts. If we run the same example with the Solidity compiler optimizer enabled, the total gas cost are lower, but the ratio is almost the same. Therefore, this optimization approach is not implemented in the optimizer.

4.2 Time-for-Space Rules

The optimization in this category are achieved by reducing the required space and instead increasing the execution time because the information has to be recalculated.

4.2.1 Time-for-Space Rule 1: Packing

Packing uses dense storage representations. There are different approaches to achieve packing. One approach that we can directly influence with the program code is overlaying. Overlaying uses the same storage space to store data that is never used at the same time.

How this can contribute to the reduction of gas cost is shown by an example:

Listing 4.5: Code example time-for-space rule 1: Packing

```
contract TimeForSpaceRule1 {
    int a; int b; int c; int d; int e; int f;

    function test(int digit) public returns(int) {
        int result = 0;
        for (int i = 0; i < digit; i++) {
            a = 10 + i;
            b = 20 + i;
            c = 30 + i;
            result += a + b + c;
        }
        d = 20;
        e = 30;
        f = 10;
        result += d + e + f;
        return result;
    }
}

contract TimeForSpaceRule1_opt {
    int a; int b; int c;

    function test(int digit) public returns(int) {
        int result = 0;
        for (int i = 0; i < digit; i++) {
            a = 10 + i;
            b = 20 + i;
            c = 30 + i;
            result += a + b + c;
        }
        a = 20;
        b = 30;
        c = 10;
        result += a + b + c;
        return result;
    }
}
```

In the first contract, we have 6 storage variables that represent integer values. In the function, we make use of all 6 variables. In the optimized version of the contract, the number of variables is reduced to 3, because they are never used at the same time.

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions with the parameter 5 through transactions are listed in Table 4.5.

The difference in the deployment cost results from the additionally required variables in the first contract. The big difference between the function calls is due to the fact that the EVM operation for setting a storage value from zero to non-zero (SSET) costs

Table 4.5: Gas cost estimate time-for-space rule 1: Packing

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	131,173	57,505	113,329	44,093
Deployment contract 2	131,045	68,505	113,265	44,093
Difference	**-128**	**0**	**-64**	**0**
Function call contract 1	148,573	127,109	144,738	123,274
Function call contract 2	89,173	67,709	85,338	63,874
Difference	**-59,400**	**-59,400**	**-59,400**	**-59,400**

20,000 gas units, while the EVM operation for resetting a storage value from non-zero to another non-zero value (SRESET) costs only 5,000 gas units.

In the example above, it would be possible to use memory variables instead of storage variables, which would be more useful in this context. But we wanted to demonstrate in an easy-to-understand example what can be achieved with the help of overlaying. This mechanism can be very useful for Solidity smart contracts to reduce gas cost, with the disadvantage of affecting the readability of the code. We conclude that the overlaying can be very useful for Solidity smart contracts. If we run the same example with the Solidity compiler optimizer enabled, the total gas cost are lower, but the ratio is almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer.

4.2.2 Time-for-Space Rule 2: Interpreters

This rule aims to reduce the space required to represent a program by using interpreters for compact representations. This can have the disadvantage of creating incomprehensible code. The simplest application of interpreter is the use of subroutines: We define part of a program in a subroutine and call it many times. How this affects the gas cost is shown by the example in Listing 4.6.

In the first contract, we define a function *test()*, which calculates a value by going through different loops. In the optimized version of the contract, instead of going through a loop five times in the function *test()*, another function *subroutine(uint256)* is introduced. The function *test()* calls the new function *subroutine(uint256)* five times.

Listing 4.6: Code example time-for-space rule 2: Interpreters

```
contract TimeForSpaceRule2 {
    function test() public returns(uint256) {
        uint256 result = 0;
        uint256 result10 = 0;
        for (uint256 i = 0; i < 10; i++) {
            result10 += i * 10;
        }
        result += result10;
        uint256 result20 = 0;
        for (uint256 i = 0; i < 10; i++) {
            result20 += i * 20;
        }
        result += result20;
        uint256 result30 = 0;
        for (uint256 i = 0; i < 10; i++) {
            result30 += i * 30;
        }
        result += result30;
        uint256 result40 = 0;
        for (uint256 i = 0; i < 10; i++) {
            result40 += i * 40;
        }
        result += result40;
        uint256 result50 = 0;
        for (uint256 i = 0; i < 10; i++) {
            result50 += i * 50;
        }
        result += result50;
        return result;
    }
}

contract TimeForSpaceRule2_opt {
    function test() public returns(uint256) {
        uint256 result = 0;
        result += subroutine(10);
        result += subroutine(20);
        result += subroutine(30);
        result += subroutine(40);
        result += subroutine(50);
        return result;
    }

    function subroutine(uint256 digit) public returns(uint256) {
        uint256 result = 0;
        for (uint256 i = 0; i < 10; i++) {
            result += i * digit;
        }
        return result;
    }
}
```

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions through transactions are listed in Table 4.15.

Table 4.6: Gas cost estimate time-for-space rule 2: Interpreters

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	156,245	77,129	134,191	59,711
Deployment contract 2	145,245	68,917	120,303	49,299
Difference	**-11,000**	**-8,212**	**-13,888**	**-10,412**
Function call contract 1	25,711	4,439	25,149	3,877
Function call contract 2	25,948	4,676	25,318	4,046
Difference	**237**	**237**	**169**	**169**

We need less gas for our optimized contract version for deployment. Regarding recurring cost, calling the function *test()* in our optimized contract version costs about 200 additional gas units. This shows us that the introduction of a subroutine does not lead to gas savings for the use of our contract due to the additional overhead for calling the function. We therefore conclude that the rule in this form generally is not applicable for Solidity smart contracts. If we run the same example with the Solidity compiler optimizer enabled, the total gas cost are lower, but the ratio is almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer, which we expected because we identified this rule as inapplicable.

4.3 Loop Rules

The optimizations of this category are achieved by changing or removing loops.

4.3.1 Loop Rule 1: Code Motion of Loops

The first loop rule describes how repeated calculations that are inside a loop and do not depend on the loop variable can be moved outward. With this help, the calculation is performed only once instead of every iteration. A small example of what this looks like in a Solidity smart contract is given in Listing 4.7.

In the first contract, we create an array and loop through it to set initial values. Each value is the index position value that is added to a costly calculated value. The costly calculated value is calculated separately for each iteration. In the optimized version of the contract, the function *someExpensiveOperation()* is not called for each value individually, but only once outside the loop.

Listing 4.7: Code example loop rule 1: Code motion of loops

```
contract Loop1 {
    function doSomething() public {
        uint256[] memory array = new uint256[](10);
        for (uint256 i = 0; i < 10; i++) {
            array[i] = i + someExpensiveOperation();
        }
    }

    function someExpensiveOperation() private returns(uint256) {
        return sqrt(199) + sqrt(234) * 212 - 234;
    }

    function sqrt(uint256 x) private returns (uint256 y) {
        uint z = (x + 1) / 2;
        y = x;
        while (z < y) {
            y = z;
            z = (x / z + z) / 2;
        }
    }
}

contract Loop1_opt {
    function doSomething() public {
        uint256[] memory array = new uint256[](10);
        uint256 expensiveValue = someExpensiveOperation();
        for (uint256 i = 0; i < 10; i++) {
            array[i] = i + expensiveValue;
        }
    }

    function someExpensiveOperation() private returns(uint256) {
        return sqrt(199) + sqrt(234) * 212 - 234;
    }

    function sqrt(uint256 x) private returns (uint256 y) {
        uint256 z = (x + 1) / 2;
        y = x;
        while (z < y) {
            y = z;
            z = (x / z + z) / 2;
        }
    }
}
```

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions through transactions are listed in Table 4.13.

The difference in the deployment cost is due to the additional variable *expensiveValue* in the optimized version. When calling the function *doSomething()*, the transaction that contains the function call for the optimized contract is much cheaper because the function

Table 4.7: Gas cost estimate loop rule 1: Code motion of loops

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	142,489	66,717	128,417	55,305
Deployment contract 2	144,097	67,917	129,961	56,505
Difference	**1,608**	**1,200**	**1,544**	**1,200**
Function call contract 1	24,634	3,362	24,423	3,151
Function call contract 2	23,274	2,002	23,129	1,857
Difference	**-1,360**	**-1,360**	**-1,294**	**-1,294**

someExpensiveOperation is called only once instead of 10 times. Of course, this continues to decrease with the number of iterations.

When the optimizer is disabled, we have to invest 1,608 more gas for deploying the optimized contract and are able to save 1,360 gas for each function call. Therefore, the additional investment for the deployment of the contract pays off after the second function call. With the optimizer enabled, we have to invest 1,544 more gas for deploying the optimized contract and are able to save 1,294 gas for each function call. Therefore, the additional investment for the deployment of the contract also pays off after the second function call.

The only case, where the optimized version requires more gas than the non-optimized version is, when the iteration count is 1. Therefore, calculations that do not depend on the loop variable should always be moved outside the loop. This can always be applied to Solidity smart contracts. Chen et al. [CLLZ17] describe a loop-related anti-pattern called *Repeated computation in a loop* that has the same motivation. If we run the same example with the Solidity compiler optimizer enabled, the total gas cost are lower, but the ratio is almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer.

4.3.2 Loop Rule 2: Combining Tests

The goal of this rule is for a loop to contain only one test or as few tests as possible. The tests should be combined so that some exit conditions can be simulated or supported by other tests. An example of how the tests can be combined is given in Listing 4.8.

In the first contract, we calculate a number i, which depends on the parameter n specified for this function. As long as n is between 1 and 10, n will be decreased and i will be incremented. In the optimized version of the contract, we move the test $n < 10$ out of the loop. The reason for this is that once the condition $n < 10$ is evaluated as true, it

does not get false in the loop, since the value n is only decreased. Once the condition evaluates to false, the loop condition is false and therefore the value i can be returned immediately.

Listing 4.8: Code example loop rule 2: Combining tests

```
contract Loop2 {
    function doSomething(uint256 n) public returns(uint256) {
        uint256 i = 0;
        while (n < 10 && n > 1) {
            i++;
            n--;
        }
        return i;
    }
}

contract Loop2_opt {
    function doSomething(uint256 n) public returns(uint256) {
        uint256 i = 0;
        if (n >= 10) {
            return i;
        }
        while (n > 1) {
            i++;
            n--;
        }
        return i;
    }
}
```

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions with the parameter value 9 through transactions are listed in Table 4.8.

Table 4.8: Gas cost estimate loop rule 2: Combining tests

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	112,793	43,693	103,879	37,087
Deployment contract 2	114,133	44,693	105,015	37,887
Difference	**1,340**	**1,000**	**1,136**	**800**
Function call contract 1	22,685	1,221	22,486	1,022
Function call contract 2	22,430	966	22,234	770
Difference	**-255**	**-255**	**-252**	**-252**

The difference in deployment cost is again due to the additional code introduced in the optimized version. The function call in the optimized version is cheaper, because we only have one condition in the loop instead of two conditions.

To calculate after how many function calls the initial investment pays off, we examine the cost when the optimizer is disabled and in the other case when the optimizer is enabled separately. When the optimizer is disabled, we have to invest 1,340 more gas for deploying the optimized contract and are able to save 255 gas for each function call. If we divide the additionally invested gas for the deployment by the gas saved for a function call, we come to the conclusion that the function must be called at least 6 times in order to achieve savings through this optimization. With the optimizer enabled, we have to invest 1,136 more gas for deploying the optimized contract and are able to save 252 gas for each function call. If we divide the additionally invested gas for the deployment by the gas saved for a function call, we come to the conclusion that the function must be called at least 5 times in order to achieve savings through this optimization.

Even if we call the function with the parameter value 10, the optimized version is cheaper, since we evaluate an if-condition instead of a loop-condition (325 gas units instead of 338 gas units for execution). Therefore, we conclude that combining tests can always be used to optimize Solidity smart contracts. If we run the same example with the Solidity compiler optimizer enabled, the total gas cost are lower, but the ratio is almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer.

4.3.3 Loop Rule 3: Loop Unrolling

The idea behind loop unrolling is to remove small loops to save the cost of modifying the loop variables and checking the loop condition. A way to apply this rule is given in the example below:

Listing 4.9: Code example loop rule 3: Loop unrolling

```
contract Loop3 {
    function sumUp(uint256[5] memory digits) public returns(uint256) {
        uint256 sum = 0;
        for (uint256 i = 0; i < 5; i++) {
            sum += digits[i];
        }
        return sum;
    }
}

contract Loop3_opt {
    function sumUp(uint256[5] memory digits) public returns(uint256) {
        return digits[0] + digits[1] + digits[2] + digits[3] + digits[4];
    }
}
```

In the first contract, we loop through an array of fixed size *digits*, which is given as a parameter, and calculate the sum of the elements it contains. In the optimized version, instead of looping through the elements, the elements are summed up directly.

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions with the array *[1,2,3,4,5]* through transactions are listed in Table 4.9.

Table 4.9: Gas cost estimate loop rule 3: Loop unrolling

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	129,093	55,905	119,499	48,699
Deployment contract 2	140,793	65,117	116,079	46,099
Difference	**11,700**	**9,212**	**-3,420**	**-2,600**
Function call contract 1	23,278	1,046	23,161	929
Function call contract 2	22,899	667	22,695	463
Difference	**-379**	**-379**	**-466**	**-466**

The difference in the deployment cost is due to the additional code introduced in the optimized version. In the optimized version, we were able to save about 400 units of gas for the function call. If the optimizer is enabled, we can even save gas for deploying the contract.

When the optimizer is disabled, we have to invest 11,700 more gas for deploying the optimized contract and are able to save 379 gas for each function call. If we divide the additionally invested gas for the deployment by the gas saved for a function call, we come to the conclusion that the function must be called at least 31 times in order to achieve savings through this optimization. Since the deployment cost are decreased when the optimizer is enabled, we conclude that this optimization strategy can still be applied on Solidity smart contracts. If we use the same example with the Solidity compiler optimizer enabled, the total gas cost for the function calls are lower, but the ratio is almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer.

4.3.4 Loop Rule 4: Transfer-Driven Loop Unrolling

This rule can be applied when trivial assignments within a loop make up a large part of the cost. Instead, the code should be repeated and the use of variables adjusted. An example of how to get rid of a trivial assignment *int j = i;* is given in Listing 4.10.

Listing 4.10: Code example loop rule 4: Transfer-driven loop unrolling

```
contract Loop4 {
    function doSomething() public returns(int) {
        int result = 0;
        for (int i = 0; i < 100; i++) {
            int j = i;
            if (i >= 20 && i <= 40) {
                i++;
            }
            if (j < 20) {
                result++;
            } else if (j > 40) {
                result += 2;
            }
        }
        return result;
    }
}

contract Loop4_opt {
    function doSomething() public returns(int) {
        int result = 0;
        for (int i = 0; i < 100; i++) {
            if (i < 20) {
                result++;
            } else if (i > 40) {
                result += 2;
            }
            if (i >= 20 && i <= 40) {
                i++;
            }
        }
        return result;
    }
}
```

In the first contract, we introduce a function *doSomething()* that contains a loop from 0 to 100. Within the loop, we introduce a new variable j, which is an alias for the loop variable i. In the second contract, we want to get rid of the trivial assignment $int\ j = i;$. To achieve this goal, all subsequent occurrences of j must be replaced by i. Since the variable i is changed in one of the conditions, we have to move the modifying condition to the bottom of the function.

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions through transactions are listed in Table 4.10.

We were able to save some gas for the deployment, because we removed an additional variable j. To perform the function call, we needed a little bit less gas.

Not only is this a loop-related optimization pattern, it can also be useful for Solidity smart contracts to save storage/memory space. Therefore, we come to the conclusion

Table 4.10: Gas cost estimate loop rule 4: Transfer-driven loop unrolling

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	118,363	47,899	111,045	42,293
Deployment contract 2	116,819	46,699	109,705	41,293
Difference	**-1,544**	**-1,200**	**-1,340**	**-1,000**
Function call contract 1	21,690	418	21,637	365
Function call contract 2	21,677	405	21,632	360
Difference	**-13**	**-13**	**-5**	**-5**

that this optimization rule can always be applied to smart contracts. If we run the same example with the Solidity compiler optimizer enabled, the total gas cost are lower, but the ratio is almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer.

4.3.5 Loop Rule 5: Unconditional Branch Removing

The idea of this rule is to remove unconditional branches at the end of a loop. Instead, the loop should be rotated in order to have a conditional branch at the end. An example of how this rule can be applied is given below:

Listing 4.11: Code example loop rule 5: Unconditional branch removing

```
contract Loop5 {
    function doSomething() public returns(int) {
        int result = 0;
        for (int i = 0; i < 10; i++) {
            result += i;
        }
        return result;
    }
}

contract Loop5_opt {
    function doSomething() public returns(int) {
        int result = 0;
        int i = 0;
        do {
            result += i;
            i++;
        } while (i < 10);
        return result;
    }
}
```

Both contracts contain a loop that goes from 0 to 10 and increments a value. The only difference between the two contracts lies in the control structure used. In the first contract, we use a for-loop, which is translated like a while loop with an introduced loop variable. In the second contract, we replace this for-loop with a do-while-loop to have the conditional jump at the end of the loop.

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions through transactions are listed in Table 4.11.

Table 4.11: Gas cost estimate loop rule 5: Unconditional branch removing

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	100,455	34,487	95,217	30,481
Deployment contract 2	99,115	33,487	93,877	29,481
Difference	**-1,340**	**-1,000**	**-1,340**	**-1,000**
Function call contract 1	22,216	944	22,038	766
Function call contract 2	22,049	777	21,871	599
Difference	**-167**	**-167**	**-167**	**-167**

By shifting the loop condition to the end of the loop, we were able to save gas for the deployment as well as for the execution of the function. The reason for this is that we have removed a conditional jump at the beginning and an unconditional jump at the end of the loop, and instead only have a conditional jump at the end.

It is not always possible to replace the for-loop/while-loop with a do-while-loop, but this should be done whenever possible. This rule is applicable for Solidity smart contracts. If we run the same example with the Solidity compiler optimizer enabled, the total gas cost are lower, but the ratio is almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer.

4.3.6 Loop Rule 6: Loop Fusion

Loop fusion is a rule that combines multiple loops that apply to the same set of elements into one. This rule was also described by Chen et al. [CLLZ17]. An example of loop fusion is given in Listing 4.12.

In the first contract, we loop through the array *digits* twice and change the result x. In the second contract, we combine both loops into one, to loop through the array only once.

Listing 4.12: Code example loop rule 6: Loop fusion

```
contract Loop6 {
    function doSomething(uint256[] memory digits) public returns(uint256) {
        uint256 x = 0;
        for (uint256 i = 0; i < digits.length; i++) {
            x += digits[i] * i;
        }
        for (uint256 i = 0; i < digits.length; i++) {
            x += digits[i] + i;
        }
        return x;
    }
}

contract Loop6_opt {
    function doSomething(uint256[] memory digits) public returns(uint256) {
        uint256 x = 0;
        for (uint256 i = 0; i < digits.length; i++) {
            x += digits[i] * i + digits[i] + i;
        }
        return x;
    }
}
```

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions with the parameter *[1,2,3,4,5,6,7,8,9,10]* through transactions are listed in Table 4.12.

Table 4.12: Gas cost estimate loop rule 6: Loop fusion

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	168,515	86,335	158,593	78,929
Deployment contract 2	159,997	79,929	150,879	73,123
Difference	**-8,518**	**-6,406**	**-7,714**	**-5,806**
Function call contract 1	26,903	3,327	26,817	3,241
Function call contract 2	26,190	2,614	26,112	2,536
Difference	**-713**	**-713**	**-705**	**-705**

We were able to save gas for deploying the contracts, because the code became shorter, and also for the execution of the function, since the loop was executed only once in the optimized version of the contract.

This rule is always applicable for Solidity smart contracts. If we run the same example with the Solidity compiler optimizer enabled, the total gas cost are lower, but the ratio is almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer.

4.4 Logic Rules

This category deals with logic evaluations that test the program state. The rules describe how the logic of the code can be modified to increase efficiency without semantic changes.

4.4.1 Logic Rule 1: Exploit Algebraic Identities

The idea of this rule is to replace expensive expressions with semantically identical cheaper expressions. We will give some small examples:

One example mentioned in the book [Ben82] is the application of the law of De Morgan:

Listing 4.13: Code example logic rule 1: De Morgan's law

```
contract Logic1 {
    bool a = false;
    bool b = false;

    function test() public returns(bool) {
        if (!a && !b) {
            return true;
        }
        return false;
    }
}

contract Logic1_opt {
    bool a = false;
    bool b = false;

    function test() public returns(bool) {
        if (!(a || b)) {
            return true;
        }
        return false;
    }
}
```

In the first contract we introduced a if-condition that checks whether a and b are false. In the second contract we applied De Morgan's law to get rid of a *NOT* operation.

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions through transactions are listed in Table 4.13.

We were able to save about 1,000 amounts of gas for deploying the optimized contract and 12 amounts of gas for each function call. Therefore, we derive that applying the De

Table 4.13: Gas cost estimate logic rule 1: De Morgan's law

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	112,227	41,871	101,829	35,505
Deployment contract 2	111,155	41,071	100,821	34,705
Difference	**-1,072**	**-800**	**-1,008**	**-800**
Function call contract 1	23,259	1,987	23,144	1,872
Function call contract 2	23,247	1,975	23,132	1,860
Difference	**-12**	**-12**	**-12**	**-12**

Morgan law is applicable for Solidity smart contracts. If we do the same thing with the Solidity compiler optimizer enabled, the total gas cost are almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer.

The book also mentions an example of how a condition can be simplified:

Listing 4.14: Code example logic rule 1: Algebraic simplification

```
function test() {
  if (sqrt(X) > 0) {
    ...
  }
}

function test_opt() {
  if (X != 0) {
    ...
  }
}
```

The expression in the first function can be replaced by a simple check, if X is not equal to zero. This is due to the square root semantics, since the square of an integer is greater than zero iff that integer is not zero. If we do the same thing with the Solidity compiler optimizer enabled, the total gas cost are lower, but the ratio is almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer.

Of course, this list of applicable examples of exploiting algebraic identity rule is not complete. To apply this rule, one has to understand the underlying algebra. The rule is applicable to Solidity smart contracts, but it is difficult to identify these parts of optimization autonomously, because they depend heavily on the use case.

4.4.2 Logic Rule 2: Short-circuiting Monotone Functions

This rule can be applied when a monotone function is tested for a threshold. The idea is to break the function as soon as the result is known, to avoid extra calculations.

A typical example of the short-circuit evaluation is that instead evaluating *A and B*, we evaluate *A cand B*. That means, if A is already true, B will not be evaluated, because the entire expression has already been evaluated to true. This can not be used if B has some important side effects. A function without side effects is marked as *pure* in Solidity.

There are also more complex applications for this rule. One example from the book [Ben82] describes a program that calculates the sum of an array and returns if that sum is greater than a cut off value. The program can look like this:

Listing 4.15: Code example logic rule 2: Short-circuiting monotone functions

```
contract Logic2 {
    function sumGtCutOff(uint256[] memory array) public returns(bool) {
        uint256 sum = 0;
        for (uint256 i = 0; i < array.length; i++) {
            sum += array[i];
        }
        return sum > 100;
    }
}

contract Logic2_opt {
    function sumGtCutOff(uint256[] memory array) public returns(bool) {
        uint256 sum = 0;
        uint256 i = 0;
        while (i < array.length && sum <= 100) {
            sum += array[i];
            i++;
        }
        return sum > 100;
    }
}
```

In the first contract, we calculate the sum of the whole array and check whether this sum is greater than 100. In the optimized version, instead of calculating the entire array, an additional loop condition check is introduced. This additional check allows us to stop the loop once the threshold is reached. If the threshold is usually reached at the top of the list, this rule is very effective. Otherwise, we will have to pay more gas for the extra condition we need to evaluate for each loop step.

This rule is generally applicable for Solidity smart contracts, but it is again difficult to autonomously detect these parts because it depends heavily on the use case.

4.4.3 Logic Rule 3: Reordering Tests

The idea of this rule is to rearrange the tests (conditions) so that the cheap tests, most likely evaluated to *true*, are placed before the expensive tests, which are most likely to be evaluated to *false*. An example for reordering tests might look like this:

Listing 4.16: Code example logic rule 3: Reordering tests

```
contract Logic3 {
    function test(int input) public returns(string memory) {
        if (input > 0) {
            return "it is positive.";
        } else if (input < 0) {
            return "it is negative.";
        } else if (input == 0) {
            return "it is zero.";
        }
    }
}

contract Logic3_opt {
    function test(int input) public returns(string memory) {
        if (input == 0) {
            return "it is zero.";
        } else if (input > 0) {
            return "is is positive.";
        } else if (input < 0) {
            return "it is negative.";
        }
    }
}
```

In this example, we check the input number to see if it is zero, negative or positive. Depending on the most common inputs, the order of conditions could be optimized. If the condition *input == 0* is most likely evaluated to *true*, our optimized version of the contract would be more gas efficient, as we would first check for equality with zero and skip all other tests.

To make this rule effective, one must know which tests are the most likely to be true. The rule is generally applicable to Solidity smart contracts, but it is also hard to recognize autonomously, as it depends heavily on the use case.

4.4.4 Logic Rule 4: Pre-compute Logical Functions

The aim of this rule is to replace the calculation of a logical function by a lookup table. This rule is related to *space-for-time rule 2: Store precomputed results*, in which we also pre-compute the values and store them in a lookup table.

Because storage operations are one of the most expensive ones, this rule does not apply to Solidity smart contracts for the same reason as space-for-time rule 2. The pre-computation may save some gas in the future, but the initial effort is far too high and access to the precomputed results is still very expensive.

4.4.5 Logic Rule 5: Boolean Variable Elimination

The idea behind this rule is to get rid of boolean variables to replace them with an *if-then-else* condition. The example in the book [Ben82] can be translated into Solidity smart contracts as follows:

Listing 4.17: Code example logic rule 5: Boolean variable elimination

```
contract Logic5 {
    function test() public {
        bool v = logicalExpression();
        doA();
        if (v) {
            doX();
        } else {
            doY();
        }
    }

    function logicalExpression() private returns(bool) {
        return true;
    }

    function doA() private { /* ... */ }
    function doX() private { /* ... */ }
    function doY() private { /* ... */ }
}

contract Logic5_opt {
    function test() public {
        if (logicalExpression()) {
            doA();
            doX();
        } else {
            doA();
            doY();
        }
    }

    function logicalExpression() private returns(bool) {
        return true;
    }

    function doA() private { /* ... */ }
    function doX() private { /* ... */ }
    function doY() private { /* ... */ }
}
```

In the first contract, we introduce a boolean variable in which we store the result of *logicalExpression()*. The book [Ben82] assumes that in the first function call, the function *doA()* must be called after the logical expression has been evaluated. Depending on the value of the boolean variable, we either execute *doX()* or *doY()*. In the optimized version of the contract, we move *logicalExpression()* to the condition and remove the boolean variable.

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions through transactions are listed in Table 4.14.

Table 4.14: Gas cost estimate logic rule 5: Boolean variable elimination

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	97,775	32,487	95,421	30,681
Deployment contract 2	97,839	32,487	89,857	26,481
Difference	**64**	**0**	**-5,564**	**-4,200**
Function call contract 1	21,521	249	21,513	241
Function call contract 2	21,508	236	21,477	205
Difference	**-13**	**-13**	**-36**	**-36**

We need a little more gas to deploy the optimized contract without the optimizer because the code was larger than the non-optimized contract code. For the execution of the function, we are able to save a small amount of gas without the boolean variable.

When the optimizer is disabled, we have to invest 64 more gas for deploying the optimized contract and are able to save 13 gas for each function call. If we divide the additionally invested gas for the deployment by the gas saved for a function call, we come to the conclusion that the function must be called at least 5 times in order to achieve savings through this optimization. Since the deployment cost are decreased when the optimizer is enabled, we conclude that even if gas savings are low, this optimization strategy can be to Solidity smart contracts.

When we run the same example with the optimizer of the Solidity compiler, the total gas cost are lower. Deploying the optimized version was cheaper than deploying the non-optimized version of the contract. The total gas cost for calling the functions is lower, but the ratio is roughly the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer.

4.5 Procedure Rules

This category deals with the underlying structure of a program organized in procedures.

4.5.1 Procedure Rule 1: Collapsing Procedure Hierarchies

Collapsing procedure hierarchies describes the mechanism by which procedure calls can be eliminated and the code can instead be inserted directly into the procedure with appropriate variable binding. The application of this rule is intended for non-recursive procedures. This rule is the counterpart to the *time-for-space rule 2: Interpreters* we have already discussed. An example is given in Listing 4.18.

Listing 4.18: Code example procedure rule 1: Collapsing procedure hierarchies

```
contract Procedure1 {
    function test() public returns(uint256) {
        uint256 result = 0;
        result += subroutine(10);
        result += subroutine(20);
        result += subroutine(30);
        result += subroutine(40);
        result += subroutine(50);
        return result;
    }

    function subroutine(uint256 digit) public returns(uint256) {
        uint256 result = 0;
        for (uint256 i = 0; i < 10; i++) {
            result += i * digit;
        }
        return result;
    }
}

contract Procedure1_opt {
    function test() public returns(uint256) {
        uint256 result = 0;
        uint256 result10 = 0;
        for (uint256 i = 0; i < 10; i++) {
            result10 += i * 10;
        }
        result += result10;
        uint256 result20 = 0;
        for (uint256 i = 0; i < 10; i++) {
            result20 += i * 20;
        }
        result += result20;
        uint256 result30 = 0;
        for (uint256 i = 0; i < 10; i++) {
            result30 += i * 30;
        }
        result += result30;
```

```
        uint256 result40 = 0;
        for (uint256 i = 0; i < 10; i++) {
            result40 += i * 40;
        }
        result += result40;
        uint256 result50 = 0;
        for (uint256 i = 0; i < 10; i++) {
            result50 += i * 50;
        }
        result += result50;
        return result;
    }
}
```

In the first contract, we define a function *test()*, which calculates a value with the help of calling a subroutine *subroutine(uint256)* five times. In the optimized version of the contract, instead of calling the helper function, the value is calculated directly by going through different loops.

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions through transactions are listed in Table 4.15.

Table 4.15: Gas cost estimate time-for-space rule 2: Interpreters

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	145,245	68,917	120,303	49,299
Deployment contract 2	156,245	77,129	134,191	59,711
Difference	**11,000**	**8,212**	**13,888**	**10,412**
Function call contract 1	25,948	4,676	25,318	4,046
Function call contract 2	25,711	4,439	25,149	3,877
Difference	**-237**	**-237**	**-169**	**-169**

The deployment actually costs, as expected from the knowledge of time-for-space rule 2, more for the optimized version, but we were able to save some gas for calling the function.

To calculate after how many function calls the initial investment pays off, we examine the cost when the optimizer is disabled and in the other case when the optimizer is enabled separately. When the optimizer is disabled, we have to invest 11,000 more gas for deploying the optimized contract and are able to save 237 gas for each function call. If we divide the additionally invested gas for the deployment by the gas saved for a function call, we come to the conclusion that the function must be called at least 47 times in order to achieve savings through this optimization. With the optimizer enabled, we have to

invest 13,888 more gas for deploying the optimized contract and are able to save 169 gas for each function call. If we divide the additionally invested gas for the deployment by the gas saved for a function call, we come to the conclusion that the function must be called at least 83 times in order to achieve savings through this optimization.

This rule therefore is generally not applicable to Solidity smart contracts. Applying this rule will also affect the readability and maintainability of the code. It might be a good idea to find a compromise between writing everything inline and put everything in separate procedures. If we do the same thing with the Solidity compiler optimizer enabled, the total gas cost are even higher. Therefore, we conclude that this optimization approach is not implemented in the optimizer.

4.5.2 Procedure Rule 2: Exploit Common Cases

This rule describes the goal of treating frequent cases efficiently and all other cases correctly. This is related to the already discussed *space-for-time rule 3: Caching*, which we found to be applicable to Solidity smart contracts. This rule can be used in Solidity smart contracts as given in Listing 4.19.

In the first contract, we take an input and call the function *calculate(int)* where we simulate some work. We assume that the most common case is input 1. In the optimized version, we have therefore changed the function *calculate(int)* that takes the *input* as an argument and decides which procedure to call, depending on the argument. If the input is the most common case 1, we call the more efficient function *specialProcedure(int)*. In all other cases, we call the function *generalProcedure(int)*.

Listing 4.19: Code example procedure rule 2: Exploit common cases

```
contract Procedure2 {
    function doSomething(int input) public {
        int x = calculate(input);

        // ...
    }

    function calculate(int input) private returns(int) {
        // simulate a lot of work
        for (int i = 0; i < 50; i++) {
            input++;
        }
        for (int i = 0; i < 50; i++) {
            input--;
        }
        return (input * 12345 / 5) - 13;
    }
}
```

```
contract Procedure2_opt {
    function doSomething(int input) public {
        int x = calculate(input);
        // ...
    }

    function calculate(int input) private returns(int) {
        if (input == 1) {
            return specialProcedure();
        }
        return generalProcedure(input);
    }

    function generalProcedure(int input) private returns(int) {
        // simulate a lot of work
        for (int i = 0; i < 50; i++) {
            input++;
        }
        for (int i = 0; i < 50; i++) {
            input--;
        }
        return (input * 12345 / 5) - 13;
    }

    function specialProcedure() private returns(int) {
        return 2456;
    }
}
```

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions with the most common case *1* in the first function call and a not so common case *5* in the second function call through transactions are listed in Table 4.16.

Without enabling the optimizer, we needed around 12,000 more gas for deploying our optimized contract due to the additional functions. Calling the functions with the most common case *1* results in about 8,000 amounts of gas that we could save. When we call the function with a not so common case *5*, we need 29,565 amounts of gas for the whole transaction, which is 70 gas more than for the non-optimized version. This overhead is due to the newly introduced condition and the additional function call.

When the optimizer is disabled, we have to invest 12,548 more gas for deploying the optimized contract and are able to save 7,697 gas for each function call of a common case. Therefore, the additional investment for the deployment of the contract pays off after the second function call of a common case. 40 additional calls with less common cases are still covered with the savings.

As long as the general procedure for calculating the less common cases is very expensive for the most common cases, this rule is applicable for Solidity smart contracts. If we run the same example with the Solidity compiler optimizer enabled, we were able to save

Table 4.16: Gas cost estimate procedure rule 2: Exploit common cases

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	121,987	50,699	109,653	121,119
Deployment contract 2	134,535	60,111	41,493	50,099
Difference	**12,548**	**9,412**	**-68,160**	**-71,020**
1. Function call contract 1	29,495	8,031	27,566	6,102
1. Function call contract 2	21,798	334	21,747	283
Difference	**-7,697**	**-7,697**	**-5,819**	**-5,819**
2. Function call contract 1	29,495	8,031	27,566	6,102
2. Function call contract 2	29,565	8,101	27,636	6,172
Difference	**70**	**70**	**70**	**70**

about 70,000 amounts of gas for deploying the optimized version of the contract. The total gas cost for calling the functions are lower, but the ratio is almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer.

4.5.3 Procedure Rule 3: Coroutines

To the best of our knowledge, the EVM does not support coroutines and therefore they can not be implemented in Solidity smart contracts. Therefore, this rule is not applicable to Solidity smart contracts.

4.5.4 Procedure Rule 4: Transformations on Recursive Procedures

This rule describes how a recursive function can be rewritten iteratively. The goal is to develop a more efficient solution with the potential disadvantage of reducing the readability of the program. We will give an example of how the Fibonacci program we have already used for the *space-for-time rules* can be optimized with this rule:

Listing 4.20: Code example procedure rule 4: Transformation of recursive procedures

```
contract Procedure4 {
    function getFibonacci(uint256 n) public returns(uint256) {
        if (n <= 1) {
            return n;
        }
        return getFibonacci(n - 1) + getFibonacci(n - 2);
    }
}
```

```
contract Procedure4_opt {
    function getFibonacci(uint256 n) public returns(uint256) {
        if(n <= 1) {
            return n;
        }

        uint256 fib = 1;
        uint256 prevFib = 1;

        for(uint256 i = 2; i < n; i++) {
            uint256 temp = fib;
            fib += prevFib;
            prevFib = temp;
        }
        return fib;
    }
}
```

The first contract implements the Fibonacci function recursively. The second contract uses a loop instead to get rid of the recursion.

Estimated gas cost for deploying these contracts in the Remix JavaScript VM and calling the corresponding functions with the parameter *15* through transactions are listed in Table 4.17.

Table 4.17: Gas cost estimate procedure rule 4: Transformation of recursive procedures

	Optimizer disabled		Optimizer enabled	
	transaction	execution	transaction	execution
Deployment contract 1	109,641	41,293	104,275	37,287
Deployment contract 2	118,707	48,299	106,157	38,693
Difference	**9,066**	**7,006**	**1,882**	**1,406**
Function call contract 1	181,508	160,044	178,500	157,036
Function call contract 2	23,015	1,551	22,513	1,049
Difference	**-158,493**	**-158,493**	**-155,987**	**-155,987**

To deploy the optimized version of the contract without enabling the optimizer, we needed about 9,000 more amounts of gas. This is due to the additional variables and code size. When calculating the Fibonacci number of *15*, we were able to save over 150,000 amounts of gas.

When the optimizer is disabled, we have to invest 9,066 more gas for deploying the optimized contract and are able to save 158,493 gas for each function call. Therefore, the additional investment for the deployment of the contract pays off after the first function

call. With the optimizer enabled, we have to invest 1,882 more gas for deploying the optimized contract and are able to save 155,987 gas for each function call. Therefore, the additional investment for the deployment of the contract pays off after the first function call.

Therefore, as in every other language, replacement of recursive functions with iterative functions should be applied to Solidity smart contracts. If we run the same example with the Solidity compiler optimizer enabled, the total gas cost for calling the function are lower, but the ratio is almost the same. Therefore, we conclude that this optimization approach is not implemented in the optimizer.

4.5.5 Procedure Rule 5: Parallelism

There is no support for concurrency in the EVM [DGHK17, SH17]. The smart contracts are always executed sequentially. For this reason, we can not use optimization techniques that involve parallelism.

4.6 Expression Rules

This category deals with expression optimizations, such as reusing results or replacing expensive expressions with cheaper ones.

4.6.1 Expression Rule 1: Compile-Time Initialization

This rule extends the previously discussed *loop rule 1: Code motion out of loops*. The idea is to initialize as many variables as possible at compile-time. This rule can be applied as follows:

Listing 4.21: Code example expression rule 1: Compile-time initialization

```
contract Expression1 {
    int constant x = 10;
    int constant y = 2;

    function test() public returns(int) {
        return x * y;
    }
}

contract Expression1_opt {
    int constant x = 10;
    int constant y = 2;

    function test() public returns(int) {
        return 20;
    }
}
```

In the first contract, we define two constants x and y. The function *test()* calculates the product of these two values. In the optimized version, the expression is replaced directly by the result, since the two values are constants.

With this optimization, we have removed the expression and save at least 5 amounts of gas for the multiplication. This rule is always applicable to Solidity smart contracts when used with constant values. If we do the same thing with the Solidity compiler optimizer enabled, the gas required to call the function of both contracts is exactly the same. Therefore, we conclude that this optimization approach is already implemented in the optimizer.

4.6.2 Expression Rule 2: Exploit Algebraic Identities

This rule has already been discussed in the *logic rule 1: Exploit algebraic identities*. Another example mentioned in the book [Ben82] is replacing *ln(A) + ln(B)* with its algebraic equivalent *ln(A * B)*.

To apply this rule, one has to understand the underlying algebra. In any case, this rule is applicable to Solidity smart contracts, but it is difficult to identify these optimization parts autonomously.

4.6.3 Expression Rule 3: Common Sub-expression Elimination

Instead of calculating the same expression several times, we save the result of the first time and reuse it every other time. This has already been discussed in the *space-for-time rule 4: Lazy evaluation*, where we stored previously calculated results in a lookup table for further requests with the same value.

This rule can always be applied if the number of accesses is large enough to justify the overhead of storing the precomputed values in a lookup table. This depends on the particular use case, but we conclude that the expression rule 3 is in general applicable to Solidity smart contracts.

4.6.4 Expression Rule 4: Pairing Computation

In this rule, the goal is to combine expressions as pairs that are similar and evaluated together, and pack them into a procedure. Pairing computation is related to *loop rule 6: Loop fusion*, where we combine several loops over the same dataset to one loop and *procedure rule 2: Exploit common cases* where we introduce a new procedure for the common cases. A sample application for this rule is finding the minimum and the maximum value of a set separately and optimize it by combining it in one procedure.

Therefore, like loop rule 6 and procedure rule 2, this rule is applicable to Solidity smart contracts.

4.6.5 Expression Rule 5: Exploit Word Parallelism

As pointed out in Section 4.5.5, there is no support for concurrency in the EVM [DGHK17, SH17]. The smart contracts are always executed sequentially. For this reason, we can not use optimization techniques that involve parallelism.

4.7 Summary

We now give a brief summary of the rules as to whether they are applicable for Solidity smart contracts or not. The list of the *Space-for-Time*, *Time-for-Space* and *Loop* rules with the decision whether they are applicable or not, together with a brief explanation, can be found in Table 4.18. The decision for the *Logic*, *Procedure* and *Expression* rules can be found in Table 4.19. "Yes" means the rule is generally applicable and not already included in the Solidity compiler optimizer. "No" means the rule is generally not applicable to Solidity smart contracts. "Mostly" means that the rule is generally applicable, but the use of the rule depends heavily on the use case and thus can hardly be optimized autonomously. "Opt" means the rule is generally applicable but has already been optimized in the Solidity compiler optimizer.

We examined a total of 27 rules and we identified 19 rules that are applicable for Solidity smart contracts. Of these 19 rules, 2 are already implemented in the Solidity compiler optimizer. 7 of these rules are not generally applicable, but in most cases.

The rules identified as "mostly applicable" can not be implemented in our prototype because they can not be autonomously optimized. This results in the following list of rules that we can implement in our prototype:

1. Loop Rule 1: Code Motion out of Loops
2. Loop Rule 2: Combining Tests
3. Loop Rule 3: Loop Unrolling
4. Loop Rule 4: Transfer-Driven Loop Unrolling
5. Loop Rule 5: Unconditional Branch Removing
6. Loop Rule 6: Loop Fusion
7. Logic Rule 1: Exploit Algebraic Identities (De Morgan's law)
8. Logic Rule 5: Boolean Variable Elimination
9. Procedure Rule 4: Transformations on Recursive Procedures
10. Time-for-Space Rule 1: Packing

Table 4.18: Summary of the Space-for-Time, Time-for-Space and Loop Rules

Space-for-Time		
Data Augmentation	No	Storage is more expensive than most operations.
Store Precomputed Results	No	Storage is very expensive, potentially wasting space and time.
Caching	Opt	Memory is much cheaper, but is already implemented in the Solidity optimizer.
Lazy Evaluation	Mostly	Can save space and time, but depends heavily on the use case.
Time-for-Space		
Packing	Yes	Storage can be reused, saving 15,000 amounts of gas per reuse.
Interpreters	No	Calling a subroutine is more expensive than writing code directly. Only makes sense if the subroutine is called very often.
Loop		
Code Motion of Loops	Yes	Repeated calculations are moved outside a loop and therefore only calculated once.
Combining Tests	Yes	Reduces the number of evaluated conditions within a loop.
Loop Unrolling	Yes	Applicable for a few iterations, but reduces the readability of the code.
Transfer-Driven Loop Unrolling	Yes	Removing trivial assignments (aliases) within a loop and replacing them with the values themselves reduces the required space.
Unconditional Branch Removing	Yes	Using a do-while loop instead of a while- or for-loop removes a conditional jump operation at the beginning of the loop.
Loop Fusion	Yes	Combining loops across the same collections to one can save computation and space.

Table 4.19: Summary of the Logic, Procedure and Expression Rules

Logic		
Exploit Algebraic Identities	Yes	We can save gas by replacing expensive expressions with semantically equivalent cheaper expressions.
Short-circuiting Monotone Functions	Mostly	We can save unnecessary calculations if we break out of the loop as soon as the result does not change.
Reordering Tests	Mostly	Check those tests that most likely evaluate to true first.
Pre-compute Logical Functions	No	Storage is very expensive, potentially wasting space and time.
Boolean Variable Elimination	Yes	Direct evaluation of the condition instead of storing the result in a boolean variable.
Procedure		
Collapsing Procedure Hierarchies	No	Deployment cost are way higher, reduces the readability and maintainability of code.
Exploit Common Cases	Mostly	When rare cases are much more expensive to calculate than common cases.
Coroutines	No	The EVM does not support coroutines.
Transformations on Recursive Procedures	Yes	Iterative algorithms should always be preferred to recursive ones.
Parallelism	No	There is no concurrency in the EVM.
Expression		
Compile-Time Initialization	Opt	Constants can always be replaced by their values.
Exploit Algebraic Identities	Mostly	As with logic rule 1, a semantically equivalent, cheaper expression should be preferred.
Common Sub-expr Elimination	Mostly	As with space-for-time rule 4, it is applicable if the number of accesses is large enough.
Pairing Computation	Mostly	Related to loop rule 6 and procedure rule 2, the combination results in gas savings because the procedure is cheap.
Exploit Word Parallelism	No	There is no concurrency in the EVM.

CHAPTER 5

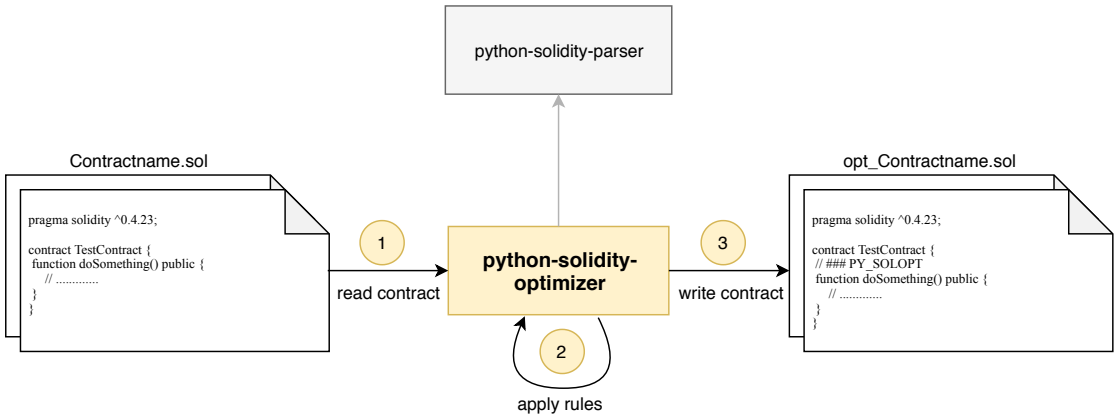
Implementation

In this chapter, we give an overview of our implemented prototype *python-solidity-optimizer*. We start by explaining the system design, the basic workflow and the dependencies our prototype relies on. Afterwards, we discuss the implemented features in order to give some insights into the development.

5.1 System Design

In this section, we describe the design, its dependency and the workflow of the prototype *python-solidity-optimizer*. A simplified illustration of the system structure is given in Figure 5.1 and is further discussed in the following sections.

Figure 5.1: System Design



5.1.1 Workflow

The basic workflow is shown in Figure 5.1. The dependency on the module *python-solidity-parser* will be described in the next section. In the first step, the prototype gets some source code files as input. Therefore, the source directory can be specified as an input parameter when starting the application. The second step is the core of the prototype. It runs through all the given source code files and analyzes these contracts according to the identified rules from Chapter 4. Some of the rules can only be identified as possible rule violations, others are automatically optimized. That part of the prototype will be discussed further in the next section. As a third and final step, an optimized version is generated for each source code file (if possible) and saved in an output directory. The output directory can again be specified as an input parameter when starting the program.

5.1.2 Dependencies

We first took a closer look at the Solidity compiler *solc*, which offers the possibility to generate an abstract syntax tree from a source file. This abstract syntax tree can be generated in JSON format for easier parsing. Since there is important information for us omitted in an abstract syntax tree [GVRB+12], we looked for a way to generate the parse tree[1] from a source file. Instead of manually analysing this syntax tree, we looked for some existing tools or modules that already wrap the solidity parser. During our research, we came across the python module *python-solidity-parser*[2] which we decided to use. This module ports the JavaScript ANTLR parser[3] to Python. The JavaScript ANTLR parser is built on top of Solidity's ANTLR4 grammar[4].

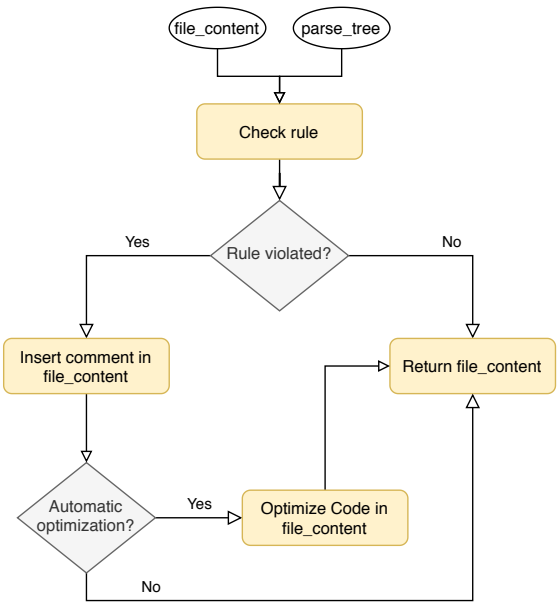
The *python-solidity-parser* module takes the source file in text form as input and generates a parse tree. This parse tree is packed into an object, so that the tree nodes can be accessed in an object-oriented manner. An important aspect for using this library was the source location information available for each node. This information would have been lost when generating an abstract syntax tree and would therefore not be sufficient for our application. The location in the source file is important, because we want to change certain parts of the source code and need the position where the new parts should be inserted or the obsolete parts should be deleted.

5.2 Prototype Implementation

In our discussion of the optimization rules, as described by Bentley [Ben82], in Chapter 4, we identified 19 out of 27 rules that are applicable to Solidity smart contracts. We also found that 10 of these rules are candidates for autonomous optimization. For this reason,

[1] Also called syntax tree [GVRB+12].
[2] `https://github.com/ConsenSys/python-solidity-parser`, accessed 24.01.2020
[3] `https://github.com/federicobond/solidity-parser-antlr`, accessed 25.01.2020
[4] `https://github.com/solidityj/solidity-antlr4`, accessed 24.01.2020

Figure 5.2: Rule Implementation



we decided to implement a prototype, for which we selected and implemented some of these optimization rules, to show a way to apply them autonomously. Our prototype is implemented in Python 3. The source code is published under the GNU General Public License 3 [gpl] and can be found on GitHub[5].

The general way in which we implemented the different rules is shown in Figure 5.2. Each rule has a function "check_rule", which takes the file content and the parsed statements as inputs. In this function, various checks are made to determine whether the rule has been violated or not. If a violation is found, we add a comment above the source line where the violation was found. Otherwise, the file content is returned unchanged. In case we implemented the automatic optimization, the optimizations are applied and the source code from the input file is modified regarding the optimization rules. The changed source code is then returned.

5.2.1 Implemented Rules

We implemented the various rules in a defensive white-listing manner, so that only known cases are optimized. With this approach, we avoid false positives with the disadvantage that we may not be able to detect all of the rule violations. The rule *Packing* is not implemented in the prototype and will be left out for future work, since the application of this rule strongly interferes with the logical flow of the function and it is difficult for the user to understand and change the code afterwards. It is also difficult to implement

[5] https://github.com/TamaraBrandstaetter/python-solidity-optimizer, accessed 26.01.2020

this rule defensively, because it generates a false positive whenever a usage of a variable is not found.

The prototype detects and automatically optimizes violations of the following rules (see Chapter 4):

- **Exploit algebraic identities**: Application of De Morgan's law.
- **Boolean variable elimination**: Direct evaluation of a condition instead of storing the result in a boolean variable.
- **Code motion of loops**: Repeated calculations are moved outside the loop and therefore only calculated once.
- **Loop unrolling for simple loops**[6]: Removing the loop and pasting the content for each iteration.
- **Transfer-driven loop unrolling**: Removing trivial assignments (aliases) within a loop and replacing them with the values themselves.
- **Unconditional branch removing**: Using a do-while loop instead of a while- or for-loop removes a conditional jump operation at the beginning of the loop.

The prototype detects violations of the following rules. Instead of an automated optimization, a note how to fix the violation or how to check whether it is a violation is inserted as a comment:

- **Combining tests**: Reduce the number of evaluated conditions within a loop in order to contain (in best case) only one condition.
- **Loop unrolling for complex loops**[7]: Removing the loop and pasting the content for each iteration.
- **Loop fusion**: Combining successive loops across the same collections to one.
- **Transformations of recursive procedures**: Iterative algorithms should always be preferred to recursive ones.

[6] The prototype detects all rule violations of that kind. Automatically optimized are simple loops that only contain expressions.

[7] Loops that contain nested loops or conditions are not automatically optimized.

5.2.2 Usage of the Prototype

The prototype accepts the following optional command line parameters:

- **-initialize**: That option is used to fetch contracts from *etherscan.io* and will be further discussed in Chapter 6 under *Data Source*.
- **-preprocess**: That option is used to preprocess contracts fetched from *etherscan.io* and will be further discussed in Chapter 6 under *Data Source*.
- **-input** [**folder**]: Sets the input folder that contains the Solidity source files. Default set to "input".
- **-output** [**folder**]: Sets the output folder for the optimized versions of the source files. Default set to "output".

CHAPTER 6

Evaluation

In this chapter, we discuss the evaluation process and the results. First, we describe the test data set that we use for the evaluation. Then, we use the implemented prototype to find rule violations in the test data set. For some of the rule violations, code optimizations are performed automatically. Finally, we discuss the results in detail.

6.1 Data Source

The website *etherscan.io*[1] is a blockchain explorer for the Ethereum blockchain. It offers the possibility to verify the source code of a smart contract[2]. Source code verification means uploading the source code together with the address to *etherscan.io*. The idea behind the verification is to create transparency and to strengthen the user's confidence in using a smart contract. The last 500 verified smart contracts can be viewed online. In addition, a CSV file can be downloaded that contains a list of all verified smart contracts that are published under an open source license. This list of open source contracts includes some popular and commonly used smart contracts, such as SafeMath[3]. The contracts of *etherscan.io* are often used as a data source for evaluations in related work [LSM19, CZC+18, NPS+17]. We have therefore decided to use the verified open source contracts from *etherscan.io* as a data source for the evaluation.

In order to access the contracts from *etherscan.io*, we implemented an initialization phase in our prototype, as shown in Figure 6.1. The initialization phase is divided into the following steps:

[1] `https://etherscan.io/`, accessed 29.01.2020

[2] `https://etherscan.io/verifyContract`, accessed 29.01.2020

[3] `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol`, accessed 29.01.2020

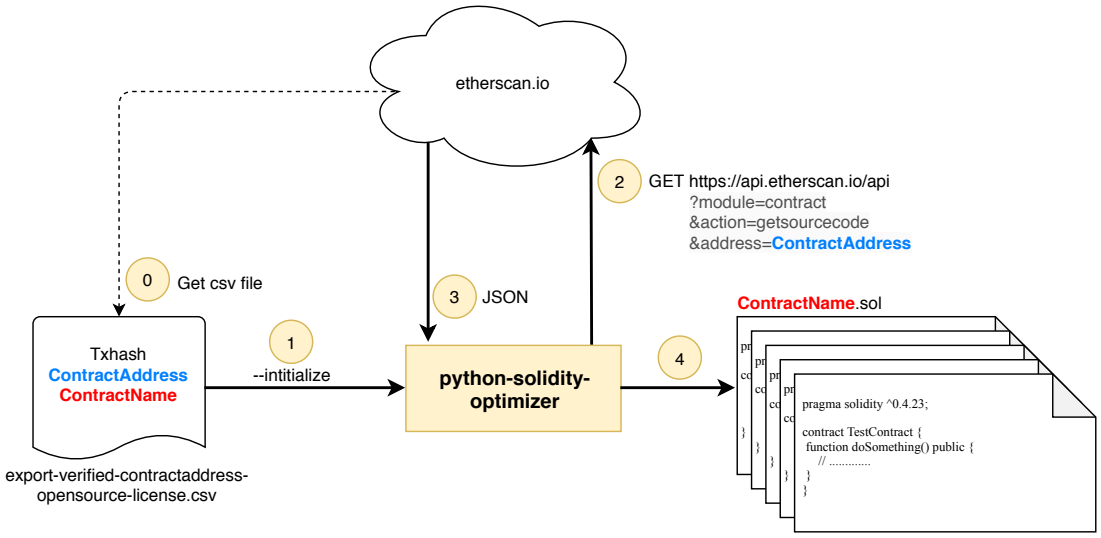


Figure 6.1: Prototype Initialization Phase

- **Step 0**: The verified contracts can be exported from *etherscan.io*[4] as a CSV file. The file contains the transaction hash (Txhash), the contract address (ContractAddress) and the contract name (ContractName) for each contract. This file is then used as input for our prototype.

- **Step 1**: The prototype *python-solidity-optimizer* can be started with an additional command line parameter *-initialize* to start the initialization phase.

- **Step 2**: For each entry in the CSV file, the prototype performs a GET-request to the REST API provided by *etherscan.io*. To get the source code of the contract, we have to specify the parameters *module=contract*, *action=getsourcecode* and *address=ContractAddress*, where *ContractAddress* is replaced by the actual address from the CSV file. Additionally, an API key has to be requested from the *etherscan.io* website and added as a parameter *apikey* for each request.

- **Step 3**: In response to the GET-request, a JSON file is returned that contains the source code along with some status information.

- **Step 4**: For each contract that is returned by the API, the prototype parses the JSON file and generates a physical file that is named after the contract name. Corrupt files that can not be parsed are omitted.

[4] `https://etherscan.io/exportData?type=open-source-contract-codes`

6.2 Evaluation

In this section, we use our implemented prototype to find the various rule violations in our test data set. The data set consists of 3,018 verified open source smart contracts from *etherscan.io*, as described in the section above.

If a contract is used within an other contract, as some kind of dependency, that particular source code is also included in the source file. Therefore, violations in the dependency contract can be counted multiple times, depending on how often they are used. Since the used dependency also has to be deployed, we can not filter out these violations, since even though the rule might be violated by another smart contract developer, the gas consumption affects the new contract too.

To calculate the actual gas savings that we achieved with the help of our optimizations, we deploy all contracts that are automatically optimized by our prototype in a test environment with the help of the Remix IDE[5]. We deploy the contracts before and after the optimization to compare the gas usages.

Since the needed gas also depends on the used parameters when deploying the contract, we use the same parameters for both contracts (i. e., optimized and not optimized) to make them comparable. The actual gas savings may vary in production, depending on the selected parameters. Some contracts in our test data set have many complex dependencies, for example on already deployed contracts on the Ethereum blockchain, where we do not have access to in our test environment. In that case, we put the parts which are affected by the optimization in new contracts and compare them isolated.

After the deployment, we call all functions that were optimized from our prototype to compare the gas usages of the function calls. Again, the needed gas depends on the used parameters. Therefore, we used the same parameters for both function calls to make them comparable. Some functions rely on randomization, which is often the case in games. We rewrite those randomized functions to return a fixed value to guarantee the same execution for the initial and the optimized version of the contract.

6.2.1 Logic Rules

The first category of rule violations we discuss are logic rules (see Section 4.4). We implemented two different logic rules, which we describe below.

Exploit Algebraic Identities

The first rule we examine is the application of De Morgan's law, as described in Section 4.4.1. This rule is referred to as Logic Rule 1 from now on.

[5] `https://remix.ethereum.org/`, accessed 18.02.2020

The prototype scans the contract for if-statements. Whenever an if-statement is found, the prototype checks the condition to find one of the following patterns:

- if (!$leftExpression$ && !$rightExpression$)
- if (!$leftExpression$ || !$rightExpression$)

These patterns are recognized as violations of the rule and replaced by the following conditions by applying De Morgan's law:

- if (!($leftExpression$ || $rightExpression$))
- if (!($leftExpression$ && $rightExpression$))

A total of 11 violations in 11 contract files were found when scanning the entire input data set according to this rule. To understand why the rule was violated only 11 times, we scanned all the if-statements in our data set. We found 31 if-statements with multiple negations. 11 of these if-statements violated our rule as described above and the optimization was performed. The remaining 20 if-statements already adhered to De Morgan's law. This means that approximately 35% of the if-statements that contain multiple negations violate our rule. A total of 2,073 if-statements with binary conditions using the operators && or || were found in our data set, that contain no negations at all.

To examine how these optimizations affect the gas cost, we deployed the initial and optimized versions of the 11 affected contracts. The total cost for deploying the contracts of this category and calling all affected functions once are shown in Table 6.1. The difference shows the gas cost savings of the optimized contract versions compared to the initial contract versions in this rule category. We were able to save overall 3,694 gas for deploying the optimized contracts. That results in an average saving of 336 gas for each contract deployment. We saved 33 gas for calling all optimized functions once. That results in an average saving of 3 gas for each function call.

Table 6.1: Gas cost savings for Logic Rule 1

	Deployment	Function call
Initial contracts	30,814,434	404,838
Optimized contracts	30,810,740	404,805
Difference	**-3,694**	**-33**

Boolean Variable Elimination

The next rule violation that we investigate is the elimination of boolean variables and evaluating the conditions directly instead, as described in Section 4.4.5. This rule is referred to as Logic Rule 2 from now on.

The prototype scans the contract for variable declarations of type *bool*. Whenever a variable declaration of this type is found, the prototype scans the remaining statements in the function to find an if-statement with an identifier as condition that is the same identifier as our previously found boolean variable.

The pattern looks like this:

```
...
bool variableName = someExpression;
...
if (variableName) {
...
}
...
```

As a last step, we check whether the boolean variable is used after the declaration, besides the if-statement. If not, we have found a violation of the rule. This means, the variable declaration can be removed and the condition replaced by the value of the boolean variable. The pattern from above is therefore replaced by the following:

```
...
...
if (someExpression) {
...
}
...
```

When scanning the entire input data set for this rule, a total of 5 violations in 5 contract files were found. To understand why the rule was only violated 5 times, we scanned all variable declarations in our data set. We found 944 boolean variable declarations. None of the remaining boolean variables fits to the pattern above, because they are used or changed afterwards and not only used as part of a condition. Due to the limitation of the usage, a violation rate of around 0.5% of all boolean variables seems appropriate to us.

To examine how these optimizations affect the gas cost, we deployed the initial and optimized versions of the 5 affected contracts. The total cost for deploying the contracts of this category and calling all affected functions once is shown in Table 6.2. The difference shows the gas cost savings of the optimized contract versions compared to the initial contract versions in this rule category. We were able to save overall 6,520 gas for deploying the optimized contracts. That results in an average saving of 1,304 gas for each contract deployment. We saved 50 gas for calling all optimized functions once. That results in an average saving of 10 gas for each function call.

Table 6.2: Gas cost savings for Logic Rule 1

	Deployment	Function call
Initial contracts	14,140,841	152,509
Optimized contracts	14,134,321	152,459
Difference	**-6,520**	**-50**

6.2.2 Loop Rules

This category deals with loop-related rule violations (see Section 4.3). We implemented six different loop rules, which we describe below.

Code Motion of Loops

This rule investigates loops to find repeated calculations that do not depend on the loop variable. These calculations are moved outside the loop and are therefore only calculated once, as described in Section 4.3.1. This rule is referred to as Loop Rule 1 from now on.

The prototype searches the contract for for-loops. Whenever a for-loop is found, the prototype searches the statements in the loop for variable declarations that are not dependent on the loop variable and are not modified within the loop. In this case, the variable declaration can be moved outside the loop. The described pattern looks like this:

```
for (int i = 0; i < 10; i++) {
  int x = someNumber * otherNumber;
  ...
}
```

After the optimization, the pattern from above would result in the following:

```
int x = someNumber * otherNumber;
for (int i = 0; i < 10; i++) {
  ...
}
```

Otherwise, if the loop variable is involved, the prototype additionally searches for pure function[6] calls to move them outside the loop. Functions that do not change the state, but are not declared with the appropriate state mutability keyword, are not recognized as a violation.

[6]By pure functions, we mean functions that do not change the state and therefore do not have to be called several times within a loop. In Solidity version 0.4.21 (`https://github.com/ethereum/solidity/releases/tag/v0.4.21`, accessed 09.02.2020), the keywords *pure* and *view* were introduced, which replace the deprecated keyword *constant*.

The described pattern looks like this:

```
for (int i = 0; i < 10; i++) {
  int x = i * someFunction(); // is a pure function
  ...
}
```

After the optimization, the pattern from above would result in the following:

```
int randomName = someFunction(); // is a pure function
for (int i = 0; i < 10; i++) {
  int x = i * randomName;
  ...
}
```

When scanning the whole input data set for this rule, a total count of 6 violations in 3 contracts were found. All 6 violations were instances of the first pattern described above. Because of the defined limitation to use only for-loops, the number is quite small.

The second described pattern of the rule could be extended to optimize all loop types in the future, since pure function calls can also exist without the dependence on loop variables. With this extension, we expect higher results in the future.

To examine how these optimizations affect the gas cost, we deployed the initial and optimized versions of the 3 affected contracts. The total cost for deploying the contracts of this category and calling all affected functions once is shown in Table 6.3. The difference shows the gas cost savings of the optimized contract versions compared to the initial contract versions in this rule category. We were able to save overall 4,102 gas for deploying the optimized contracts. That results in an average saving of 1,367 gas for each contract deployment. We saved 578 gas for calling all optimized functions once. Four different functions were affected by the optimization, since one of the functions violated the rule three times. That results in an average saving of 144 gas for each function call.

Table 6.3: Gas cost savings for Loop Rule 1

	Deployment	Function call
Initial contracts	2,082,519	570,469
Optimized contracts	2,078,417	569,891
Difference	**-4,102**	**-578**

Combining Tests

The rule combining tests aims to reduce the number of evaluated conditions within a loop in order to contain (in the best case) only one condition, as described in Section 4.3.2. From now on, this rule is referred to as Loop Rule 2.

The prototype searches the contract for all types of loops: for-loops, while-loops and do-while-loops. Whenever a loop is found, the prototype checks the loop condition, to determine whether it is an binary operation that is combined by the AND-operator (&&) or the OR-operator (||). In this case, a comment is added above the affected loop to inform the developer that the tests should be combined within the condition, if possible.

A total of 81 violations in 52 contract files were found when scanning the entire input data set according to this rule. This means that 0.4% of all loops are affected by this rule.

To understand, why the number is very low, we calculated some additional metrics. We found a total of 3,943 for-loops in our data set, while 13 violations of Loop Rule 2 come from this loop type. We found a total of 508 while-loops in our data set, while 66 violations of Loop Rule 2 come from this loop type. The remaining 2 violations come from the 18 do-while-loops we found in our data set. The analysis shows that there are most likely multiple tests within a while-loop and about 13% of all while-loops within our data set violate Loop Rule 2.

Since the optimization is not done autonomously, the gas savings can not be calculated.

Loop Unrolling

This rule aims to remove the loop and insert the content for each iteration, as described in Section 4.3.3. The rule is referred to as Loop Rule 3 from now on.

The prototype searches the contract for for-loops. We define the for-loop parts as the following:

- for (*initialization*; *condition*; *expression*) { ... }

Whenever a for-loop is found, the prototype starts checking the *initialization*. For-loops without an *initialization* are ignored and therefore not recognized as instances of this rule. To find out how often the loop is executed, the initial value in the *initialization* is stored. Next, the prototype checks the *expression*. To simplify matters, only loops are taken into account whose *expression* contains unary operations (++ or --). As a final step, the prototype checks the *condition* to find out how many loop runs would be performed. The following patterns for the loop *condition* are considered for this rule:

- variable < limit
- variable <= limit
- variable > limit
- variable >= limit

The limit must be a number for the prototype to determine the number of loop runs.

We configured our prototype to automatically unroll loops that run a maximum of 4 times for our test data set.

With this configuration, we were able to redeem the additionally invested deployment cost after a maximum of 200 function calls[7]. Unrolling loops with higher loop iterations led to higher initial deployment cost, which increased the needed function calls to 300. Therefore, if the loop is executed more often, the loop is not automatically unrolled, but a note is added as a comment above the loop, so that the developer can decide whether the loop should be unrolled manually or not. Loops that contain nested loops or conditions are also not automatically optimized, since the unrolling would again lead to higher deployment cost. The comment is also added above the loop, so that the developer can still decide whether the loop should be unrolled manually or not.

A total of 134 occurrences in 98 different contract files were found when scanning the input data set for this rule. This means approximately 3% of all loops are affected by this rule. The result contains all loops that have a fixed number of loop iterations, with the exception of 4 loops, which we filtered out due to an empty loop body. 4 of these violations were automatically optimized under the conditions described above. The remaining 130 rule violations are marked with a comment, but the developer has to check manually and decide whether the loop should be unrolled or not.

To examine how these optimizations affect the gas cost, we deployed the initial and optimized versions of the 4 affected contracts. The total cost for deploying the contracts of this category and calling all affected functions once is shown in Table 6.4. The difference shows the gas cost savings of the optimized contract versions compared to the initial contract versions in this rule category. We had to invest overall 174,363 more gas for deploying the optimized contracts. That results in an average additional investment of 43,591 gas for each contract deployment. We saved 1,682 gas for calling all optimized functions once. That results in an average saving of 421 gas for each function call. We were able to redeem the additionally invested deployment cost after 104 calls on average.

Table 6.4: Gas cost savings for Loop Rule 3

	Deployment	Function call
Initial contracts	3,654,299	3,828,662
Optimized contracts	176,443	174,761
Difference	**174,363**	**-1,682**

[7]We chose the limit of 200, since the optimizer of the Solidity compiler also optimizes functions with the condition that they are called at least 200 times.

Transfer-driven Loop Unrolling

This rule removes trivial assignments (aliases) within a loop of the loop variable and replaces them with the loop variable itself, as described in Section 4.3.4. From now on, the rule is referred to as Loop Rule 4.

The prototype searches the contract for for-loops. Whenever a for-loop is found, the prototype goes through the loop-statements and looks for variable declarations of the following pattern:

- *type alias = loopVariable;*

When a variable declaration of this pattern is found, the prototype checks the remaining loop statements to see if the variable *alias* is changed after the declaration. If the variable is not changed, there is a rule violation. In this case, a note for the developer in form of a comment is added above the loop.

When we scanned the input data set, we found one violation of this rule. Because of the limitation that we only take aliases of the loop variable into account, we did not expect many violations. In the future, this rule could be extended to recognize all aliases within a loop, without the limitation that the target must be the loop variable. With this expansion, we expect higher results in the future.

To examine how this optimization affects the gas cost, we deployed the initial and optimized version of the affected contract. The total cost for deploying the contract and calling the affected function once is shown in Table 6.5. The difference shows the gas cost savings of the optimized contract version compared to the initial contract version. We were able to save 2,208 gas for deploying the optimized contract. We saved 50 gas for calling the optimized function once.

Table 6.5: Gas cost savings for Loop Rule 4

	Deployment	Function call
Initial contract	481,510	30,569
Optimized contract	479,302	30,519
Difference	**-2,208**	**-50**

Unconditional Branch Removing

The idea behind this rule is to use a do-while loop instead of a while- or for-loop to remove a conditional jump operation at the beginning of the loop, as described in Section 4.3.5. The rule is referred to as Loop Rule 5 from now on.

In our prototype, we only implemented this rule for simple for-loops, since we have to decide whether the loop is executed at least once in order to replace the loop with

a do-while loop. A simple for-loop, as has already been defined for the Loop Rule 3, consists of an initialization that introduces a new variable with a start value, a condition with one of the operators $<$, $<=$, $>$ *or* $>=$ and an unary expression (++ or --).

The prototype scans the contract for for-loops. Whenever a for-loop is found, the prototype checks whether the loop is executed at least once. The pattern described looks like this:

```
for (int i = 0; i < 10; i++) {
  ...
}
```

After the optimization, the pattern from above would result in the following:

```
int i = 0;
do {
  ...
  i++;
} while (i < 10);
```

A total count of 138 occurrences in 98 different contract files were found when scanning the input data set for this rule. The result contains all loops that have a fixed number of loop iterations and run at least once. We found out that only 0.4% of all loops used in our data set were originally do-while-loops. Because of this, we expected some violations to occur in this category, because do-while-loops are not used very often. About 3% of all loops broke Loop Rule 5.

To examine how these optimizations affect the gas cost, we deployed the initial and optimized versions of the 98 affected contracts. The total cost for deploying the contracts of this category and calling all affected functions once is shown in Table 6.6. The difference shows the gas cost savings of the optimized contract versions compared to the initial contract versions in this rule category. We were able to save overall 126,599 gas for deploying the optimized contracts. That results in an average saving of 1,292 gas for each contract deployment. We saved 16,071 gas for calling all optimized functions once. 125 different functions were affected by the optimization. That results in an average saving of 129 gas for each function call.

Table 6.6: Gas cost savings for Loop Rule 5

	Deployment	Function call
Initial contracts	204,034,849	4,420,417
Optimized contracts	203,908,250	4,404,346
Difference	**-126,599**	**-16,071**

Loop Fusion

The purpose of this rule is to combine successive loops over the same collection into one, as described in Section 4.3.6. The rule is referred to as Loop Rule 6 from now on.

The prototype scans the contract for for-loops. Whenever a for-loop is found, the prototype checks whether the next statement is also a for-loop. If that is the case, both loops are compared naïvely. That is, the loop lines are compared as strings to check if they are the same. The pattern described looks like the following:

```
...
for (int i = 0; i < numbers; i++) {
  ...
}
for (int i = 0; i < numbers; i++) {
  ...
}
...
```

If the loops are identical, the second loop can be removed and the content of both loops should be merged. After the optimization, the pattern from above would result in the following:

```
...
for (int i = 0; i < numbers; i++) {
  ...
  ...
}
...
```

The optimization does not take place automatically, since each variable has to be checked across both loops to see whether it is even possible to merge them without changing the semantics of the program. Instead, a note for the developer in form of a comment is added that the loops should be merged into one to save gas.

A total of 10 violations in 10 different contract files were found when scanning the input data set for this rule. We did not expect many results, because looping over the same collection is rarely used. In the future, this rule could be extended to not only find two consecutive loops, but to scan all loops within a function to find possible loop reductions.

Since the optimization is not done autonomously, the gas savings can not be calculated.

6.2.3 Procedure Rules

The last category of rule violations that we are discussing are procedure rules (see Section 4.5). This category contains one rule that we describe below.

Transformations of Recursive Procedures

This rule aims to convert recursive algorithms into semantically identical iterative algorithms, as described in Section 4.5.4. The rule is now referred to as Procedure Rule 1.

The prototype scans the contract for functions. For each function, the prototype checks all statements within a function to determine whether it contains a function call with the same function name and the same number of parameters. In this case, a possible recursive function is found and a note in form of a comment for the developer is inserted above the function.

A total of 85 violations in 44 different contract files were found when scanning the input data set for this rule. Since it is possible to define several functions with the same name and the same number of parameters, but different parameter types, within one contract, this can lead to false positives. The developer must check this manually, since the matching of actual parameters to formal parameters is not implemented in the prototype.

Since the optimization is not done autonomously, the gas savings can not be calculated.

6.3 Summary

An overview of the rule violations discussed can be found in Figure 6.2. Loop Rule 5 is the most common one, because it affects all loops that have a fixed number of loop iterations and are executed at least once. Loop Rule 4 only occurs once in our data set, since only one contract uses an alias of the loop variable within a loop without editing the loop variable itself or the alias. In total, we found 471 violations across the test data set in our analysis.

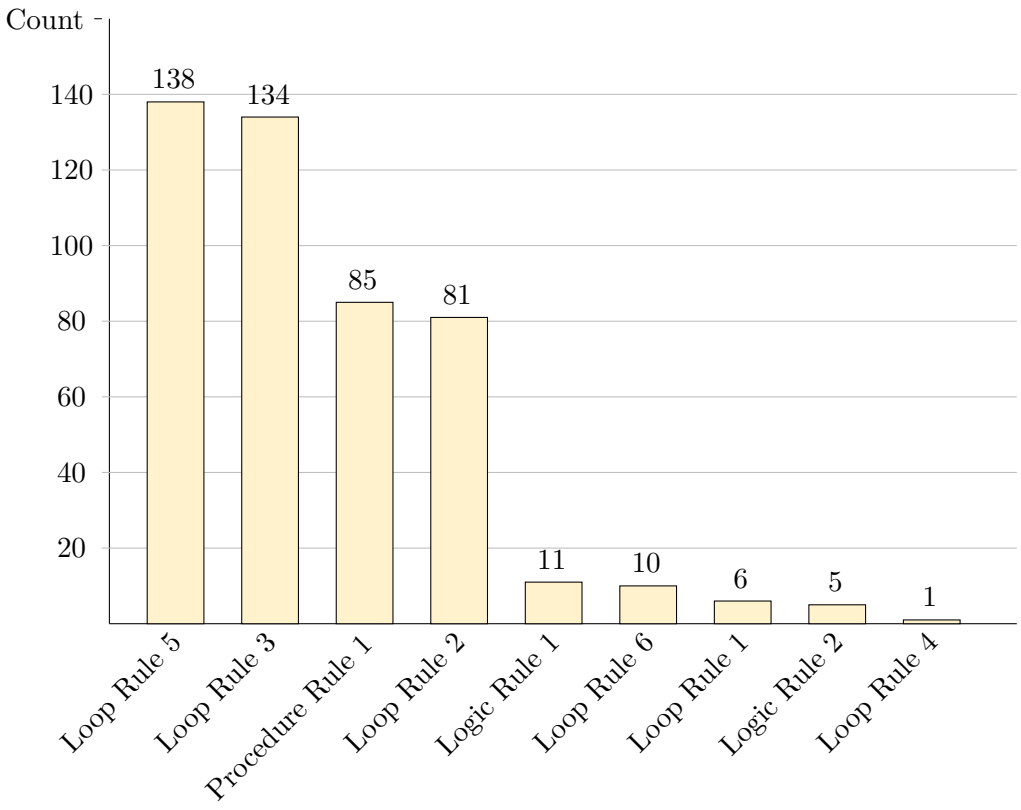


Figure 6.2: Rule Violation Count in *etherscan.io* Contracts

Those contracts that contain a violation, tend to contain multiple violations. One file contains 12 violations. The most common case is that two violations occur in the same file. The 471 violations are spread across 204 input files from the data set. Our data set consists of 3,018 contract files. That gives us a violation rate of about 6.76%. As has already been mentioned above, the open source contracts include some popular and commonly used smart contracts, such as *SafeMath*, which we did not expect to violate our rules many times. A higher violation rate is expected when analysing contracts from developers who have less experience in developing smart contracts.

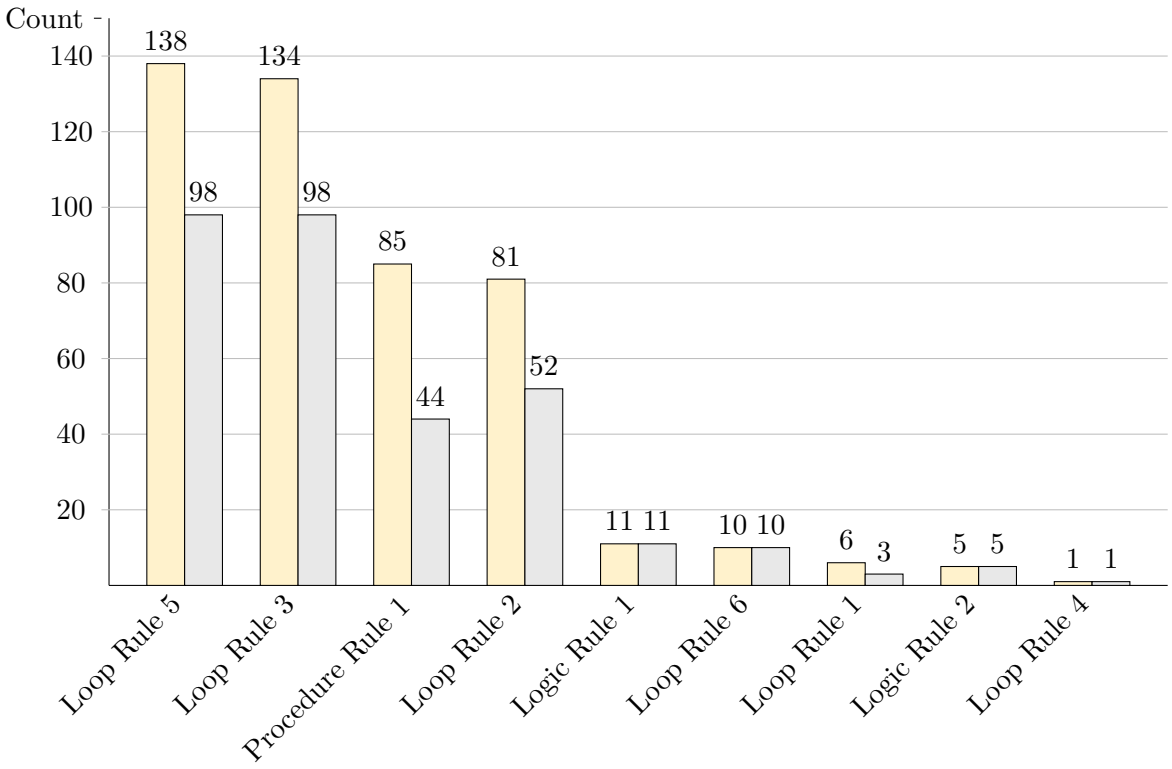


Figure 6.3: Distribution of Rule Violations to Contracts

The distribution of rule violations across the various contract files is shown in Figure 6.3. The number of violations within a rule is shown in yellow. The number of affected contract files in which the violations occur, is shown in gray.

We also analyzed how the autonomously performed optimizations affect the gas cost. The gas cost savings grouped by the rules is shown in Table 6.7. We calculated the average savings per contract for the deployment and the average savings per function call. Table 6.8 shows the savings in percent. Due to the additional investment required for Loop Rule 3, the percentage is negative.

The highest average deployment cost saving was achieved by Loop Rule 4, which was optimized once in our test data set. We were able to save 2,208 gas for the deployment. Loop Rule 3 does not lead to gas savings for the deployment, since unrolling the loop means additional produced code. The average investment for deploying a contract from Loop Rule 3 is 43,591 gas. On the other hand, these contracts of Loop Rule 3 also achieved the highest average function call cost saving. We were able to save on average 421 gas for a single function call. The lowest average function call cost saving was achieved by Logic Rule 1, which is the application of De Morgan's law. Since the optimized version needs one less NOT-Operation, which is worth 3 gas, we were able to save 3 gas for an optimized function call.

Table 6.7: Gas cost overview grouped by rules

	Deployment cost		Function call cost	
	Total	**Average**	**Total**	**Average**
Logic Rule 1 (initial)	30,814,434	2,801,312	404,838	36,803
Logic Rule 1 (optimized)	30,810,740	2,800,976	404,805	36,800
Difference	**-3,694**	**-336**	**-33**	**-3**
Logic Rule 2 (initial)	14,140,841	2,828,168	152,509	30,502
Logic Rule 2 (optimized)	14,134,321	2,826,864	152,459	30,492
Difference	**-6,520**	**-1,304**	**-50**	**-10**
Loop Rule 1 (initial)	2,082,519	694,173	570,469	142,617
Loop Rule 1 (optimized)	2,078,417	692,806	569,891	142,473
Difference	**-4,102**	**-1,367**	**-578**	**-144**
Loop Rule 3 (initial)	3,654,299	913,575	176,443	44,111
Loop Rule 3 (optimized)	3,828,662	957,166	174,761	43,690
Difference	**174,363**	**43,591**	**-1,682**	**-421**
Loop Rule 4 (initial)	481,510	481,510	30,569	30,569
Loop Rule 4 (optimized)	479,302	479,302	30,519	30,519
Difference	**-2,208**	**-2,208**	**-50**	**-50**
Loop Rule 5 (initial)	204,034,849	2,081,988	4,420,417	35,363
Loop Rule 5 (optimized)	203,908,250	2,080,696	4,404,346	35,235
Difference	**-126,599**	**-1,292**	**-16,071**	**-129**

The average gas cost savings for the deployment are illustrated in Figure 6.4. Since Loop Rule 3 requires additional investment for the deployment, the rule is omitted in the Figure. The savings for calling a function are illustrated in Figure 6.5.

Table 6.8: Gas cost savings in percent grouped by rules

	Deployment	Function call
Logic Rule 1	0.012%	0.008%
Logic Rule 2	0.046%	0.033%
Loop Rule 1	0.197%	0.101%
Loop Rule 3	-4.771%	0.953%
Loop Rule 4	0.459%	0.164%
Loop Rule 5	0.062%	0.364%

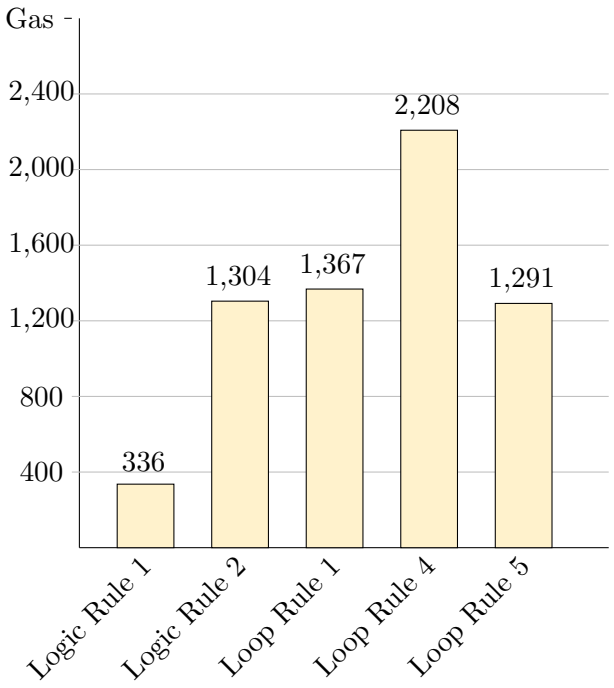


Figure 6.4: Average Gas Savings for Deployment

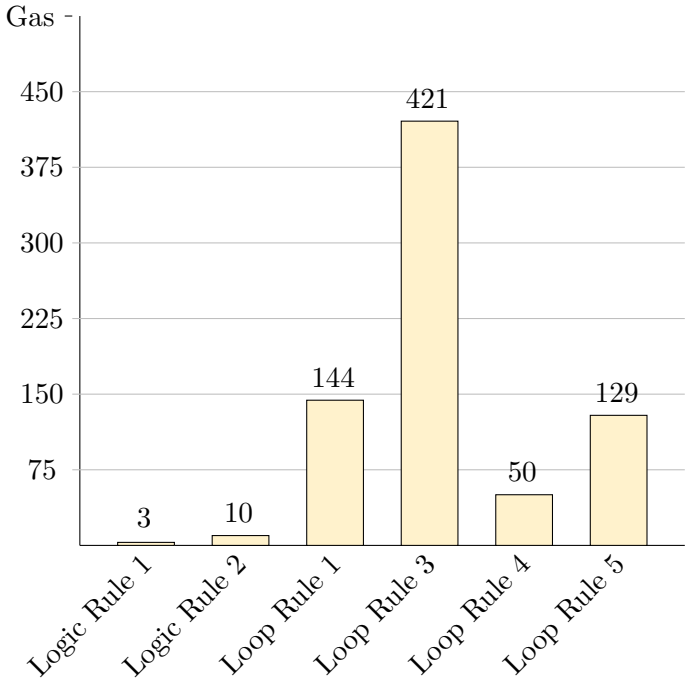


Figure 6.5: Average Gas Savings for Function Calls

CHAPTER 7

Conclusion

In this chapter, we give an overview of the whole work and summarize our findings. Afterwards, we give an outlook how the prototype can be extended and improved in the future.

7.1 Discussion

In this thesis, we examined how existing optimization strategies from the field of software engineering can be applied to Solidity smart contracts in order to improve the gas consumption. Therefore, we analyzed 27 existing optimization strategies by Bentley [Ben82], whether they lead to gas savings when applied to Solidity smart contracts or not. We analyzed the effect on the gas consumption with the latest Solidity compiler version 0.5.13 at the time of writing. We identified 19 of these strategies in form of rules as applicable for Solidity smart contracts. 2 of these rules are already implemented in the optimizer of the Solidity compiler and therefore not further considered. We also found that 10 of these rules are candidates for autonomous optimization. The remaining 7 rules highly depend on the specific use case and can not be optimized autonomously.

Therefore, we selected some of these rules and implemented a prototype to show a way to apply them autonomously. The prototype detects 9 rules whether they are violated in Solidity source code. Additionally, 6 of these rules are automatically optimized in the source code when a violation is found. Since some optimizations could interfere with the semantics of the program, we implemented the rule detection in a defensive white-listing manner. This means, only known patterns are recognized as rule violations and optimized automatically. With the white-listing approach, we tried to avoid false positives with the drawback that we may not detect all rule violations. Whenever a rule violation is found, a comment is added above the affected source code line for the developer. The idea is that the developer should always be able to decide whether the optimization is desired or

not. Since this is a prototype that has been developed to demonstrate functionality, the next section will discuss suggestions for future improvements.

During the evaluation, we analyzed 3,018 open source smart contracts from *etherscan.io*[1] with the help of our prototype. Overall, we found 471 rule violations in our test data set. These rule violations were spread across 204 different contract files. That results in a violation rate of roundabout 6.8%. For those rules, where we implemented automatic optimization, we additionally deployed the affected contracts before and after the optimization in a test environment, in order to compare the gas usages. We were on average able to save 1,213 gas for deploying an optimized contract version compared to the initial contract version and 123 gas for calling an optimized function once.

7.2 Future Work

Since the prototype has been developed to demonstrate functionality, we have some suggestions for improvements in the future, which we discuss below.

Rule Detection

We implemented the prototype in a defensive white-listing manner, which means we only recognize some known violation patterns of the rules. This part could be improved in order to recognize more rule violation patterns. Therefore, a manual analysis of existing smart contracts to find new patterns is necessary.

9 of 10 rules are detected by our prototype. We did not implement the remaining rule, *Packing*, for simplicity, since this rule strongly interferes with the logical flow of the function. In the future, we will also implement the detection of this remaining rule.

The rule *code motion of loops* finds repeated calculations inside a for-loop that do not depend on the loop variable and moves them outside the loop. Due to that definition, the rule is limited to for-loops. This could be extended in the future, to also find pure function calls in while-loops and do-while-loops.

The rule *transfer-driven loop unrolling* removes aliases of the loop variable within a loop and replaces them with the loop variable itself. Due to that definition, the rule is limited to aliases of the loop variable and therefore to for-loops. In the future, this rule could be extended to recognize all aliases within all loop types.

Automatic Optimization

At the moment, the prototype optimizes 6 out of 10 rules automatically. In the future, we want to implement the optimization for all remaining rules.

Since the prototype has been implemented to demonstrate functionality, no main focus was placed on the runtime of the prototype itself. This can be improved in the future.

[1] `https://etherscan.io/contractsVerified`, accessed 20.02.2020

Testing

The test data set from the evaluation consists of open-source smart contracts from *etherscan.io*[2]. Some popular smart contracts are included in this data source, which we did not expect to violate the rules many times. We expect a higher violation rate when testing contracts from a different data source, where the developers might have less experience in developing smart contracts.

User Interface

To make it easier for the developer to see the optimizations and to decided whether this optimization is desired or not, we want to develop a user interface in the future. It could be included, for example, in the Remix IDE[3].

[2] `https://etherscan.io/contractsVerified`, accessed 20.02.2020
[3] `https://remix.ethereum.org`, accessed 20.02.2020

List of Figures

List of Tables

Listings

Bibliography

[AGL+18] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. Ethir: A framework for high-level analysis of ethereum bytecode. In *International Symposium on Automated Technology for Verification and Analysis*, pages 513–520. Springer, 2018.

[AH74] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[AKM+15] Frederik Armknecht, Ghassan O Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. In *International Conference on Trust and Trustworthy Computing*, pages 163–180. Springer, 2015.

[B+14] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[Ben82] Jon Louis Bentley. *Writing Efficient Programs*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[CLLZ17] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, pages 442–446. IEEE, 2017.

[CLZ+18] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. Towards saving money in using smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results*, pages 81–84. IEEE, 2018.

[CZC+18] Weili Chen, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. Detecting ponzi schemes on ethereum: Towards healthier blockchain technology. In *2018 World Wide Web Conference*, pages 1409–1418. International World Wide Web Conferences Steering Committee, 2018.

[Dan17] Chris Dannen. *Introducing Ethereum and Solidity*. Springer, 2017.

[dAS19] Monika di Angelo and Gernot Salzer. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures*, pages 69–78. IEEE, 2019.

[DGHK17] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *ACM Symposium on Principles of Distributed Computing*, pages 303–312. ACM, 2017.

[Gam95] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Prentice Hall, 1995.

[gpl] Gnu general public license. `http://www.gnu.org/licenses/gpl.html`.

[GVRB+12] Dick Grune, Kees Van Reeuwijk, Henri E Bal, Ceriel JH Jacobs, and Koen Langendoen. *Modern compiler design*. Springer Science & Business Media, 2012.

[Hug89] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[Huk18] Taneli Hukkinen. Reducing blockchain transaction costs in a distributed energy market application. Master's thesis, Aalto University Finland, School of Science, 2018.

[LCO+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[Lee11] Charles Lee. Litecoin. `https://litecoin.org/`, 2011.

[LSM19] Shlomi Linoy, Natalia Stakhanova, and Alina Matyukhina. Exploring ethereum's blockchain anonymity using smart contract code attribution. 10 2019.

[Mil97] Mark S Miller. Computer security as the future of law. In *Extro 3 Conference*, 1997.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *White Paper*, 2008.

[NBF+16] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and cryptocurrency technologies: A comprehensive introduction*. Princeton University Press, 2016.

[NPS+17] Robert Norvill, Beltran Borja Fiz Pontiveros, Radu State, Irfan Awan, and Andrea Cullen. Automated labeling of unknown contracts in ethereum. In *2017 26th International Conference on Computer Communication and Networks*, pages 1–6. IEEE, 2017.

[SH17] Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In *International Conference on Financial Cryptography and Data Security*, pages 478–493. Springer, 2017.

[Sig18] Christopher Signer. Gas cost analysis for ethereum smart contracts. Master's thesis, ETH Zurich, Department of Computer Science, 2018.

[sol] Solidity documentation. `https://solidity.readthedocs.io/en/v0.5.12`.

[Sza97] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[W+14] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151, 2014.

[XWS19] Xiwei Xu, Ingo Weber, and Mark Staples. Varieties of blockchains. In *Architecture for Blockchain Applications*, pages 45–59. Springer, 2019.

[Zoh15] Aviv Zohar. Bitcoin: under the hood. *Communications of the ACM*, 58(9):104–113, 2015.