TÉCNICO LISBOA

Formal Analysis of Ethereum Virtual Machine Bytecode Patterns

Bernardo Gastão Varanda da Silva Prates

Thesis to obtain the Master of Science Degree in

Mathematics and Applications

December 2019

Acknowledgments

I would like to express my gratitude to my thesis advisors, Professors Paulo Mateus and Pedro Adão, for the guidance in the choice of the subject of this work and on the process of its development, and for the opportunity they have provided me to work on the project.

This work was partially supported by Programa Operacional Competitividade e Internacionalização, Programa Operacional Regional de Lisboa, Project Bloch, POCI-01-0247-FEDER-033823.

It warms my heart to remember all the support and joyfulness I received from my friends, specially, João Amado, who gave valuable advice for the completion of this work.

Thank you Diana, for your love, support and all the time spent beside me throughout the thesis.

I would to address a special thank you to my family, and, in particular, my parents and my grandparents, for the love, support, and constant encouragement I always received from them throughout my studies.

A final word of appreciation for everyone I did not mention and in some way may have given a contribution for this work.

Resumo

O Ethereum ganhou notoriedade pela implementação de protocolos computacionais, smart contracts. O sistema assenta no processamento de transações, que constituem um meio de executar estes contratos de modo determinístico.

Ataques do tipo *Denial-of-Service* são prevenidos através da implementação do mecanismo de gas, que pondera trabalho computacional realizado. A cryptomoeda Ether, que está embebida no sistema do Ethereum, é utilizada na compra de gas necessário às transações. Deste modo, o foco deste trabalho consiste na identificação e substituição de padrões de bytecode dispendiosos, que permite reduzir o custo associado às transações.

A ferramenta desenvolvida nesse sentido percorre o código dos contratos e substitui os padrões a cada nova ocorrência. Este procedimento é repetido até não ser possível realizar mais optimizações.

De forma a manter a integridade do contrato, a ferramenta Mythrill é usada na obtenção do grafo de controlo de fluxo, que determina se o contrato é elegível para optimização.

Embora a Máquina Virtual do Ethereum (EVM) não seja completa à Turing, a caracterização do comportamento de contratos é uma tarefa árdua. Provamos que introduzindo modificações como o aumento da capacidade de armazenamento, a EVM passa a ser completa à Turing e aferir os endereços de alvo das instruções de *jump* torna-se um problema indecidível.

Palavras-chave: Ethereum, Smart Contracts, Semântica Operacional, Completude à Turing, Optimização de Código, Contexto de Execução

Abstract

Ethereum became noticeable for implementing computerized protocols, smart contracts. As a transaction based system it is able to execute these contracts, by means of transactions, in a deterministic fashion and to keep the registers associated with these actions available publicly.

Preventing the system against abuses such as the Denial-of-Service attacks is carried out by gas, a measure of computational work. Ether, the cryptocurrency embedded in the system, is used to purchase the amount of gas necessary to allow transactions to proceed. Therefore, the identification and substitution of gas-costly bytecode patterns may reduce the cost of transactions, and shall be the focus of this work to prove that the alternative patterns constitute indeed an optimization, saving gas.

The developed tool parses the contracts' instructions and replaces costly patterns as they are found. This procedure is repeated until there are no more optimizations to be done. In order to maintain code integrity, Mythril is used to obtain the control flow graph of the contract, which determines if any optimizations may be applied.

Although the Ethereum Virtual Machine (EVM) is not Turing Complete, characterizing the behavior of contracts is still a difficult task. We prove that with few modifications, targeting mainly in terms of storage capacity, the EVM may be labeled as Turing complete and that to determine if a given jump instruction goes to a given code location is undecidable.

Keywords: Ethereum, Smart Contracts, Operational Semantics, Turing Completeness, Code Optimization, Execution Context

Contents

List of Figures

Nomenclature

Array, List and Stack Operations

$\alpha + +\beta$	concatenation of lists, stacks or arrays α and β
$\alpha.a$	accessing component a of the tuple α
$\alpha[b \rightarrow c]$, $\alpha[b- = x]$, $\alpha[b+ = x]$	substitution of the current instance for c, subtraction (addition) of the current value by x given the element b of the tuple α, respectively
$a :: \alpha$	concatenation of element a and the stack α
A^*	Kleene Closure of A
$v[i, j]$	array containing the entries $[v(i), \ldots, v(j-1)]$ of the vector v

Sets

$\mathbb{S}$	Call Stack Set	
$\mathcal{A}$	Account Set	
$\mathcal{B}$	Block Contents Set	
$\mathbb{A}$	Addresses Set	$\{0,1\}^{160}$
$\mathbb{B}$	Set of Bits	$\{0,1\}$
$\mathbb{B}^x$	Sequences of bits of length x	$\{0,1\}^x$
$\mathbb{N}_x$	Set of non-negative integers representable by x bits	
$\mathbb{T}$	Transaction Set	
$\mathbb{T}_{env}$	Transaction Environment Set	
Σ	Global States Set	
I	Execution Environments Set	
M	Machine States Set	
N	Transaction Effects Set	

Ethereum Virtual Machine

η	Transaction Effects
Γ	Transaction Environment
ι	Execution Environment
μ	Machine State
σ	Global State

List of Equations

Chapter 1

Introduction

The lack of trust deposited in the financial and economical industries combined with its cost-inefficient and rigid infrastructure, allied to the urge for a new computing model based on decentralization and collaboration led to emergence of blockchain systems.

Bitcoin, proposed in 2008 by Satoshi Nakamoto [34] and launched in 2009, was the first decentralized cryptocurrency. In contrast with Bitcoin, Ethereum, presented in 2014 [44] [9] and launched in 2015, reaches beyond implementing a merely "trustless" transaction system as it can function as a trusted platform to perform sequential program computations, referred to as smart contracts. In fact, it is considered the main platform for smart contract execution [43].

The Bitcoin ledger records all transactions that have occurred in the system until any given moment. Therefore, a transition system appears to be an adequate abstraction, with the state at each moment consisting of a collection or set of all unspent bitcoins with respective ownership (UTXO, unspent transaction outputs).

In addition to a public key, a UTXO can be owned by a script, which must be given a specific input so that one may be able to claim its ownership. However, the scripting language supported by Bitcoin is fairly limited. Recursion, loops and calls are not included in the spectrum of scripts that may be written [9].

Among the main areas where Ethereum smart contracts enjoy the most popularity are Finance and Games, in light of the wide range of applications and perks known, such as implementing tokens and polls, ability to impose call restrictions, particularly regarding caller addresses and time constraints.[43]

Smart contracts for Ethereum are usually written in Solidity, a high-level imperative programming language, although other languages like Vyper are also used, and then compiled into bytecode to be executed by the Ethereum Virtual Machine (EVM), the environment where all transaction-related computations are performed.

Comparatively to Bitcoin, in Ethereum, Solidity allows the construction of while loops, thus, making Ethereum smart contracts more manageable to write.

In the same fashion as it occurs in other blockchain based cryptocurrency systems, the blocks in Ethereum consist of a collection of ordered transactions and are appended to the ledger through a proof of work consensus operated by the miners, which have access to the state of the system.

In order to perform a transaction, the sender must pay a fee upfront in Ether, the cryptocurrency sponsored by Ethereum. Ether is converted to gas, a unit of computational work performed by the miners, at the ratio provided by the sender (Wei/gas, Wei is equivalent to 10^{-18} Ether).

Ether holds a considerable economical value (worth $\$1359.48$ in January, 14, 2018, according to [4]). In accordance, the safety and security of smart contracts is an important subject discussed by the Ethereum community. In effect, the hardships associated to the task of scrutinizing the low-level bytecode programs the contracts are compiled to, which run on the Ethereum Virtual Machine, may pose a barrier to a thorough analysis of smart contracts.

Another subject of interest in the Ethereum community deals with the optimization of gas costs, which will be the case of this thesis.

1.1 Goals and Achievements

With the knowledge of the Ethereum Operational Semantics, the main goal of this work resides in providing a formal reasoning about equivalence between sequences of instructions for the Ethereum Virtual Machine.

Bearing this in mind, the presented work is divided into two phases:

1. Proof of preservation of the effect induced by the original bytecode sequence.
2. Maintaining functionalities of the contract, analyzing the control flow corresponding to its bytecode instructions.

The Ethereum Virtual Machine bytecode language contains a structure identical in purpose to the $go\ to$, embodied in the instructions JUMP and JUMPDEST. Furthermore, conditionals are also a part of the structures one is able to write via the JUMPI/JUMPDEST pair.

Following up and describing the behavior of a program is a difficult task, due to the existence of both these structures, as the $go\ to$ is considered an harmful pattern [17].

In order to accomplish this task, we will use symbolic execution tool to generate the control flow graph of a contract.

1.2 Literature Review

The introduction to the structure of the Ethereum system is based on the White Paper by Vitalik Buterin [9], an overview article over blockchain technology and smart contracts [43], and multiple works and documents related to the Ethereum blockchain.

The definition of the semantics for the EVM follow the Yellow Paper by Gavin Wood [44] and a formalization of the EVM for the F* proof assistant provided in [23], which was completed and corrected in [45].

Optimizations applied to Ethereum smart contracts constitute a trending topic in the Ethereum community. As an example, one can look at [36], where Solidity costly patterns are evaluated, or [11] and

[10], for a perspective of bytecode optimizations. In fact, this work will discuss in further detail and under the semantic description provided by [23] and [45] the execution of the patterns presented in [11] along with other patterns found by inspecting several contracts.

Towards proving the Turing Completeness of a modified version of the EVM, the version of the Unlimited Register Machine presented by Nigel Cutland [14] will be the main reference. Several well known results on the subject of Computability presented by Cutland shall be used in the proofs of undecidability of two decision problems.

1.3 Outline

First, we provide an overview of Ethereum and refer several projects which approach the same topics discussed throughout this work.

The second chapter presents a description of the EVM small-step semantics, the notions of context of an instruction sequence evaluation, semantic equivalence between bytecode sequences, and the optimization relation. Each EVM bytecode pattern presented is given the respective optimized sequence and the proof said optimization holds.

On the third chapter it is sketched a proof of Turing completeness of a modified version of the EVM, which differs from the original definition mainly by ignoring the gas restrictions on computations and storage limits, by changing the word size and altering storage to be infinitely expansible. Additionally, two decision problems relative to the control flow modified EVM programs are proven undecidable.

Finally, the last chapter deals with the structure of smart contracts, in terms of sequential instructions, and demonstrates the main issues regarding the construction of the control flow graph of a contract.

Chapter 2

Ethereum Overview and Related Work

2.1 Ethereum Overview

Labeled as the "World Computer", Ethereum consists of a platform for performing deterministic computations in a decentralized fashion, free of third-party interference.

Decentralization is achieved through the establishment of a consensus between the nodes of the network responsible for altering the state of the system.

Although many of the concepts behind Ethereum mimic previously achieved technological breakthroughs, such as the notion of blockchain, proof-of-work, cryptocurrency, among others, the embedding of a "quasi-Turing complete" virtual machine, which is recreated by each node when processing transactions, along with the notion of system state as a collection of accounts and respective contents, distinguishes Ethereum from other blockchain based systems such as Bitcoin.

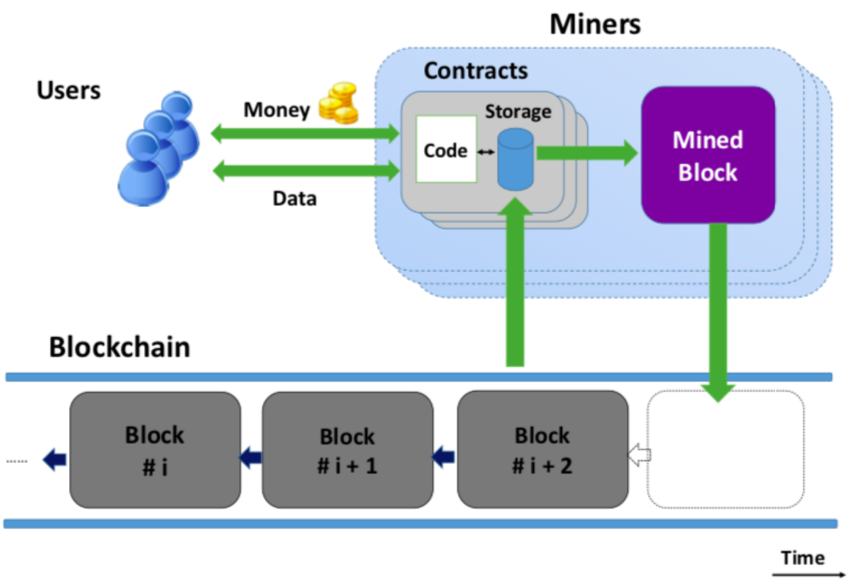


Figure 2.1: Schematics for a blockchain for execution of smart contracts [15]

2.1.1 Storage

The data stored in Ethereum is divided in two categories - permanent and ephemeral - as an example, the registers regarding transactions are stored permanently, as for the account information it is in constant evolution, due to account interaction through message calls and contract creation.

Storage is accomplished by the use of a data structure named Merkle Patricia tree, also designated by Trie, whose concept derived from the ideas incorporated in the Patricia trees and in the Merkle trees.

Regarding the first, it is a type of tree that falls into the category of prefix trees, i.e., nodes in the same branch share a common prefix. As for the latter, it consists of a binary tree, storing the data in the leaf nodes, with each parent node holding the hash of the addition of its children.

Merging these two concepts, as the Merkle tree aims for a secure way to store data, while the Patricia tree facilitates search, produces an efficient and prone to verification state information storage.

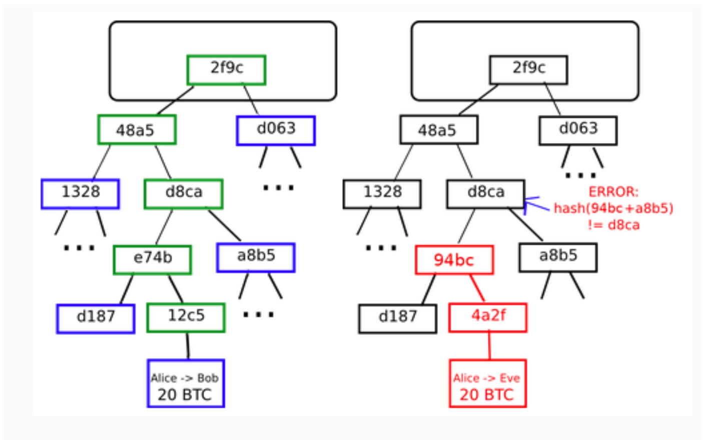


Figure 2.2: Merkle tree example from [9]

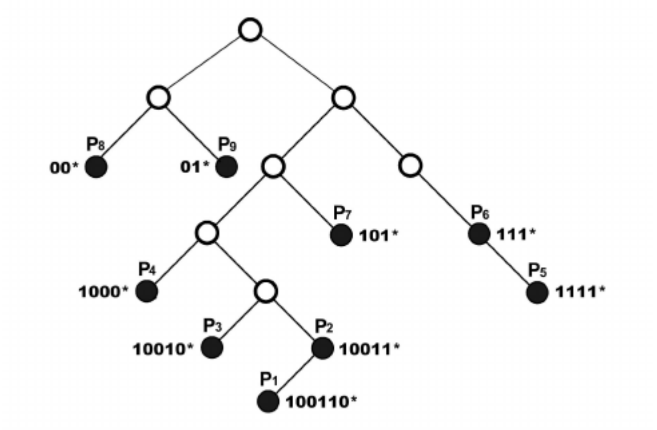


Figure 2.3: Patricia tree example from [28]

Gathering all the information generated by this system is achieved by four types of databases:

- *State Trie* contains the up-to-date information of all active accounts in the Ethereum network.
- *Storage Trie* pertains to a single account, containing the respective information.

- *Transaction Trie* is associated to each block, being filled with information relative to the values, data and participants of transactions.
- *Receipts Trie* registers the associated information with transaction logs.

In what concerns the transaction receipts, an additional data structure is used, the Bloom filter. Its goal is to accelerate search through storage of log information derived from contract execution, encoding the identification of the account responsible for producing the message and the content of the message.

The search for an element stored on this type of data structure is achieved through a probabilistic procedure, issuing no false negatives.

2.1.2 Accounts

Accounts in Ethereum are divided into two types, externally owned accounts and contract accounts. Externally owned accounts are controlled by a public and private key pair, while contract accounts obey the associated code embedded at the time of creation. These public and private keys are the product of an elliptic curve digital signature algorithm.

A contract's code is immutable, being removed from the blockchain in the event that the instruction SELFDESTRUCT is present in the code and a successful transaction issued with the contract as a recipient executes it.

Regarding the account address, it is created with resort to the Keccak-256 hash function and the public key generated for that account, for the case of externally owned accounts. Contract accounts obtain the address using the nonce of the contract creation transaction and the address of the transaction sender, i.e., the contract creator.

As previously referred, the Keccak-256 hash function is used throughout the Ethereum protocol. Moreover, the virtual machine includes an instruction or opcode, SHA3 corresponding to the Keccak-256 hash function presented for the SHA-3 (Secure hash algorithm) competition held by the US National Institute of Standards and Technology (NIST) [2]. Despite the name of the opcode SHA3, the current SHA-3 corresponds to an altered version of Keccak-256 (FIPS 202).

2.1.3 Transactions

Changes in the system state are the work of transactions. These interactions between accounts fall into three different categories - Ether transfers, contract creations, and message calls.

The payment for transactions is performed prior to execution. A mechanism to overcome fluctuations on the Ether price consists of using a measurement unit for the computational power required, called gas.

A user is required to specify the gas limit to be expended and the conversion fee to Wei at the moment of issuing the transaction. Accordingly, the correspondent amount of Wei is withdrawn from the user's account. For the case of a successful transaction or occurrence of an exceptional halting due to motives other than gas availability, the remaining gas is converted with the same fee to Wei and deposited in the users account.

A transaction is a message issued from an externally owned account either to another account (ether message call) or without recipient (contract creation). As results of a transaction, contracts are able to trigger internal transactions, interacting with other contracts and externally owned accounts. With the goal of protecting the system against Denial of Service (DoS) attacks, the number of calls deriving from a transaction is limited (1024 calls per transaction).

The failure of an internal transaction does not imply an identical result for the original transaction itself.

Valid transactions are granted the possibility of altering the system state, while exceptional halting only conduces to Ether depletion on the sender's account.

2.1.4 Blocks

The evolution of a blockchain is accomplished by the nodes of the network (miners) through successively appending blocks to it, a process designated by mining.

Blocks are structures containing a collection of transactions, an identification number, and a nonce, a numerical value necessary for block validation. The block header, which contains all the information regarding the executed transactions, also points to the previous block and ommer blocks.

Ommer blocks or uncle blocks appear as a product of the fast generation of blocks in Ethereum. The transactions contained in these blocks does not influence the state of the system, as ommer blocks are appended to side chains, instead of the main chain.

In order to encourage mining, the miners are rewarded for each uncle block mined.

The adopted strategy for determining the main chain of blocks, which corresponds to the Ethereum blockchain, the GHOST (Greedy Heaviest Observed Subtree) protocol is the common practice.

Similarly to transactions, the computational work corresponding to a given block is bounded by a gas consumption limit.

Furthermore, the restriction imposed by the gas limit restricts the volume of computations that the subsequent blocks may perform, i.e., the gas Limit of the next block may only exceed the previous in a given fraction (lesser than $\frac{1}{1024}$), according to the specification provided in the Yellow Paper[44].

Mining in Ethereum is achieved by a mechanism of the type: Proof-of-Work (PoW), a computationally expensive task which results in the inclusion of the nonce in the block.

Ethash is the algorithm employed in the Ethereum PoW. It is designed to be ASIC-resistant, meaning miners equipped with specific hardware targeting the algorithm do not possess an advantage, thus, contributing to decentralization. The main steps for the mining algorithm are described in [1].

1. Generate a seed from the block headers in the chain up until the current state.

2. From the seed, one can compute a 16 MB pseudorandom cache. Light clients store the cache.

3. From the cache, we can generate a 1 GB dataset, with the property that each item in the dataset depends only on a small number of items from the cache. Full clients and miners store the dataset, that grows linearly with time.

4. Mining involves grabbing random slices of the dataset and hashing them together. Verification can be done with low memory by using the cache to regenerate the specific pieces of the dataset that you need, so you only need to store the cache.

The large dataset referred in 3 is updated every 30000 blocks.

Checking the validity of a block is obtained through the procedure described by the steps below, which are present in the Ethereum White Paper [9].

1. Check validity of the previous block.
2. Check if the block timestamp is greater than the previous block and less than 15 minutes into the future.
3. Check PoW of the block.
4. With $\mathrm{S}[0]$ being the system state ulterior to the block, and TX, the collection of n transactions associated to the block:
   (a) For $i = 0, \ldots, n-1$, $\mathrm{S}[i+1] = \mathrm{APPLY}(\mathrm{S}[i], \mathrm{TX})$
   (b) If any applications returns an error, or if the total gas consumed in the block up until this point exceeds the $\mathrm{GASLIMIT}$, return an error.
5. $\mathrm{S}_{\mathrm{FINAL}}$ is obtained from $\mathrm{S}[n]$ by rewarding the miner.
6. Check if the Merkle Patricia tree root of the state $\mathrm{S}_{\mathrm{FINAL}}$ is equal to the final state root provided in the block header. If it is, the block is valid; otherwise, it is not valid.

Due to system modifications agreed upon by the miners or to attacks such as the DAO attack, the chain may suffer a transformation called fork [6].

In the case of the attack to the DAO contract, a fundraising contract holding approximately 150 million US dollars at the time of the attack, an attacker exploited a bug in the code, managing to place under her control 60 million US dollars. As a consequence, in order to maintain the confidence of investors, a majority of the miners operated a hard fork on the chain, nullifying the alterations of the transaction relative to the attack in a parallel chain, the current Ethereum blockchain.

Nevertheless, this measure was not consensual and miners who did not agree upon the alterations suggested maintaining the original chain.

2.1.5 Ethereum Virtual Machine

The Ethereum Virtual Machine consists of the model for the execution of all computations related to transactions.

Regarding its structure, the EVM is a Harvard-Architecture stack-based virtual machine, i.e., by opposition to the Von Neumann paradigm, the instructions to execute do not intersect with memory spaces addressed by the machine for operating on data.

As for the memory structures on which the EVM operates, they are five and may be enumerated as: the code, the input data, the stack, the memory and the account storage.

Concerning the pair code and input data, it consists of two byte arrays. The first corresponds to a sequence of bytes representing EVM instructions and, for the case of PUSH instructions, the respective argument. As for the later, it can be either an empty array, leading to invocation of the contract fallback function when defined, or an array containing the function hash in the first four bytes subsequently followed by the input parameters.

The execution stack, which may be referred to in a more simple manner as stack, by contrast with the other stack structure in Ethereum, the call stack, whose content is composed of message calls, pertains to the last in first out category, and stores 256-bit words, with no gas cost associated. Although the maximum capacity of the stack is fixed at 1024 elements, only the first 17 can be accessed directly through the execution of EVM instructions.

In the same way that the stack contents are linked to a single transaction, the memory, a 32-byte byte-array, expansible until the capacity of 2^{256} bytes, behaves as a volatile memory for the contract execution storage.

At the beginning of each message call the execution memory is empty and expansion is induced by either accessing or writing to an index greater than the memory current size.

A contract storage space is categorized as permanent storage and consists of a mapping between 256-bit words, i.e., the number of slots in an account storage is of 2^{256}, with each slot containing a 256-bit word.

In what concerns the EVM bytecode, it constitutes a low-level programming language containing 145 instructions. Even though most of the instructions consist of operations meddling with stack manipulation, a more detailed description of the categories the opcodes fall into would be as stated below.

- STOP and Arithmetic operations.
- Comparison and bitwise logic operations.
- Block and Transaction environment information.
- SHA3, which computes the Keccak-256 hash.
- Stack operations (POP, PUSH, DUP and SWAP).
- Memory and storage manipulation and control flow operations.
- System operations (related to calls and halting).

Memory manipulation is operated by the commands MSTORE, MSTORE8 and MLOAD , with the first two loading a 32-byte word and a one byte word, respectively, and the latter loads a 32-byte word from a given index. In the same fashion, storage is altered or accessed by SSTORE and SLOAD, which store/load a 256-bit word from/to a given index.

A call can be categorized into two types according to the EVM instruction that originated it. If the opcode corresponds to CALL, the sender changes to the contract that was being executed, otherwise if DELEGATECALL was issued, the agent performing the role of the sender of this internal transaction matches the sender of the original transaction and the value is unaltered.

Arithmetic operations are performed by the instructions ADD, MUL, SUB, DIV, MOD, EXP, SDIV, SMOD, ADDMOD, MULMOD and SIGNEXTEND. Every operation is restricted to modulo 2^{256}, unless it is specified by the MOD opcodes, which take the argument for the respective modulo operations.

2.2 Related Work

2.2.1 Smart Contracts' Analysis Tools

Several perspectives may be applied to the code analysis of a program and description of its behavior.

Static analysis relies on the inspection of the code without performing any execution. It is a safe method as no malicious code may be executed during the analysis.

Among the approaches that classify as static, disassembling consists of a simple procedure which is responsible for transforming binary code into a readable representation.

Determining the control flow of a program is also a technique employed in the static analysis of a program.

Symbolic execution consists of performing the execution of programs with inputs that may be symbolic. It is based on an extension of the programming language semantics, so that the language operators accept this sort of inputs.

Given the economic value of smart contracts allied with the constraints imposed by intrinsic characteristics of Ethereum and the limitations and flaws of Solidity, several tools which perform the security and safety analysis of smart contracts have been conceived.

Categorizing tools that target the analysis of smart contracts can be achieved by the following classification:

- Type of approach - static or dynamic.
- Object of study - solidity code, bytecode or decompiled bytecode.
- Focus of the analysis - structure description, call integrity, reentrancy, overflows due to arithmetic, among other topics.

As an example of an EVM bytecode decompiler, Porosity, introduced in [39], consists of a tool focused on translating the contract associated bytecode to a compatible Solidity readable code, allowing also the disassembling of bytecode into blocks.

Operating on both solidity source code and compiled code is a task managed by the SmartInspect tool in [8], with the intention of preventing recompilation by the user.

In what concerns the analysis of Ethereum bytecode exclusively, Oyente is a powerful tool that makes use of symbolic execution and the use of a Satisfiability Modulo Theories (SMT) solver to determine path feasibility.

Soundness and completeness do not figure in the main goals of Oyente, as it abstracts the EVM execution in a simplified manner [22]. Nevertheless, it was a starting point for several other projects with a wide spectrum of objectives.

The Maian tool also consists of another extension of Oyente, whose target focuses on security vulnerabilities related to multiple transactions [6].

These contributions constitute an important step in the construction of trustworthy and properly specified smart contracts.

2.2.2 EVM Semantics

A first semantic definition of the EVM was elaborated by the Ethereum co-founder Gavin Wood in the Ethereum Yellow Paper [44].

The need for a more clear introduction of some concepts and the advantage that an approach that targets theorem provers may bring in terms of increasing the safety of smart contracts lead to the works of Grishchenko *et al.* in [23] and Hildenbrandt *et al.* in [25].

In both works, the authors specify the semantics for the EVM resorting to languages linked to theorem provers - F^* and $\mathbb{K}$, respectively.

The work of Hirai detailed in [26] also establishes the bridge between smart contracts and theorem provers such as Isabelle and Coq.

A model for smart contracts as a finite state machine is proposed by Mavridou and Laszka [30]. This model is translated to Solidity, providing secure code patterns, among other security functionalities.

The semantic rules provided in [23] are perfected by Xavier in [45]. A symbolic EVM written in the Mathematica language was also implemented by the author.

2.2.3 Optimization of Smart Contracts

Gas expenditure is associated with two main issues:

- Is it possible to effectively measure gas consumption of a transaction beforehand?
- Can smart contracts be optimized in terms of gas expenditure?

A gas estimation for each function may be obtained with the Solidity compiler. However, this estimation is only possible for functions with no loops or external calls (in this case the returned value is infinite).

The Solidity compiler is also equipped with an optimizer [3], which operates by:

1. Dividing the instructions of a contract into blocks;
2. Generating expressions regarding stack, memory and storage from the block instructions and linking this information to the block.
3. Building a control flow graph and a dependency graph.
4. Eliminating the operations which are not part of the dependency graph.

Ethir, a tool which is built incorporating the Oyente (makes use of the symbolic execution feature in order to generate the control flow graph), contains a module - SACO - that operates as a static analyzer on the contract's bytecode to infer upper bounds on the number of iterations of loops.

GASTAP is also an example of a tool which resorts to the symbolic execution provided by Oyente in order to build a control flow graph . Ultimately, its goal is to estimate an upper bound on the gas consumption of each function of a given contract [5].

Beatriz Xavier also develops a tool to estimate the gas consumption of functions of contracts, provided all the function loops are bounded, resorting to the already mentioned implementation of the EVM.

Another solution for the problem of gas estimation is proposed by the MadMax tool [21]. It operates on an Intermediate Representation generated by a decompiler and uses static program analysis techniques to automatically detect gas vulnerabilities.

Gasper, announced in [10], deals with expensive and superfluous Solidity patterns and also recurs to Oyente for the generation of the control flow graph of a contract.

Analyzing the gas consumption associated with a contract's execution, is also a task performed in order to Denial-of-Service exceptions due to lack of gas, as we can observe when using the GasFuzz tool [29].

Chapter 3

EVM Semantics and Optimization

3.1 EVM Small-Step and Operational Semantics

Defining the formal semantics for a programming language is a task that can be achieved through the application of one of three paradigms:

- Denotational - the meaning of expressions is determined by a mathematical model that assigns to each language construct the corresponding effect. Therefore, semantics defined in this fashion constitute an "effect" based approach.

- Axiomatic - program specification is performed by assertions that represent properties holding before and after execution.

- Operational - constructs are defined by the computations, steps or transitions they induce on the machine. The machine can be an abstract transition machine with its transitions corresponding to state changes, or it can be modeled by axioms and rules that dictate the transitions (the latter is called structural operational semantics).

  As a consequence, the operational approach is more focused on the "process" that generates a given effect.

Structural operational semantics gives rise to a framework that describes a computation in individual steps. For this reason, it is sometimes referred to as small-step Semantics.

The EVM semantics description presented by Grishchenko *et al.* [23], corrected and completed by Xavier in [45], will guide this work, providing the structure to reason formally about the execution of EVM bytecode patterns. The paradigm adopted by these authors corresponds to the small-step semantics.

3.1.1 Syntax

Before characterizing the transition relation that can describe the execution of contracts by the EVM stepwise, the elements of an execution state and remaining concepts regarding execution of the EVM instructions must be specified. Furthermore, the notation to be applied is nearly identical to the one presented in work by Xavier [45].

Accounts, Transactions and Blocks

Definition 1. *An account is represented by the tuple,* $(n, b, stor, code)$*, where* n *is the nonce - the number of contract creations made by the account,* b *stands for the value of the account balance in Wei,* $stor$ *corresponds to the account's persistent repository (a mapping between words in* $\mathbb{B}^{256}$*) and* $code$ *stands for the bytecode of the contract (which can be empty, in case of externally owned accounts).*

As a consequence the accounts set is defined as $\mathcal{A} = \{(n, b, stor, code) | \; n, b \in \mathbb{N}_{256}, \; stor \in \mathbb{B}^{256} \to \mathbb{B}^{256}, \; code \in (\mathbb{B}^8)^*\}$*.*

A transaction is an essential element in the Ethereum system, as all state transitions are transaction based. A formal definition of a transaction corresponds to:

Definition 2. *A transaction is a tuple,* $T = (from, to, n, v, d, g, p)$*, with an address corresponding to the sender -* $from$*, the recipient -* to*, the nonce of the sender -* n*, the amount of Wei to transfer -* v*, the input data -* d*, the gas limit -* g *and the gas price -* p*. Under this definition, the transactions set is thus defined as* $\mathbb{T} = \mathbb{A} \times \mathbb{A} \times \mathbb{N}_{256} \times \mathbb{N}_{256} \times (\mathbb{B}^8)^* \times \mathbb{N}_{256} \times \mathbb{N}_{256}$

Blocks are structures associated with lists of transactions, $\mathcal{T}$, and form a sequence representing the changes that occur in the system.

Additionally to the transaction list, blocks store:

- the nonce, a security parameter obtained through the PoW algorithm, being associated to the system state and the node which mined the block;
- the block identifier;
- the gas limit allowed for the computations of all transactions on the list;
- the address of the miner (beneficiary's address);
- the difficulty associated with mining the block;

Definition 3. *A block* B *is a tuple* $(nonce, \mathcal{T}, i, l, ben, d)$ *containing:*

- *nonce -* $nonce \in \mathbb{B}^8$*;*
- *collection of transactions -* $\mathcal{T} = (T_1, \ldots, T_m), \; T_i \in \mathbb{T}$ *for* $i = 1, \ldots, m$*;*
- *number which identifies the block -* $i \in \mathbb{N}_{256}$*.*
- *gas limit -* $l \in \mathbb{N}_{256}$*.*
- *beneficiary's address -* $ben \in \mathbb{A}$*;*
- *difficulty -* $d \in \mathbb{N}$*.*

The set of blocks is denoted by $\mathcal{B} = \mathbb{B}^8 \times \mathbb{T}^* \times \mathbb{N}_{256} \times \mathbb{N}_{256} \times \mathbb{A} \times \mathbb{N}$.

The definition of the Ethereum blockchain is not consensual, as the chain was forked multiple times since the genesis block. Nevertheless, the sequence referred by the following definition points to what is considered to be the main active Ethereum blockchain.

Definition 4. *The Ethereum blockchain is the sequence of blocks $B = (B_0, \ldots, B_k)$, with B_0 corresponding to the genesis block -* $(42, \emptyset, 0, 3141592, 0, 2^{17})$ *- and B_k to the most recently mined block.*

Execution States

An element of the execution state which is already introduced in the Ethereum Yellow Paper [44] consisting of information pertaining to all of the accounts in the system (the state data base abstraction) is the global state, also referred as world state.

Definition 5. *The global state consists of a partial mapping between addresses and accounts, $\sigma \in \mathbb{A} \to \mathcal{A}$. The set of global states is denoted by Σ.*

The global state of an account a is denoted by $\sigma(a) = (b, code, stor, n)$.

The empty account consists of the tuple $(0, \varepsilon, 0_F, 0)$, with 0_F denoting the constant function whose value is always zero. Moreover, the representation of an empty or non-existing account associated to the address a is achieved by the symbol $\perp$, i.e., $\sigma(a) = \perp$.

Considering that the global state is defined as a mapping, the update of a given account's components is processed in a different manner than the change in an element of a tuple. Accordingly, $\sigma\langle a \to \sigma(a)[\mathsf{c} \to x]\rangle$ denotes the change of component c of $\sigma(a)$ to the value x.

Three additional elements describe the execution of EVM instructions: machine state, execution environment and transaction effects.

Definition 6. *The machine state, $\mu = (gas, pc, m, i, s, rd)$, is the tuple holding information such as the remaining gas available to conclude the transaction, the program counter - pc, a byte array containing a set of words stored, the memory, and the number of active ones - m and i, respectively, the stack contents - s, and the return data - rd.*

The set of machine states is denoted by $M = \mathbb{N}_{256} \times \mathbb{N}_{256} \times (\mathbb{B}^8)^* \times \mathbb{N}_{256} \times ((\mathbb{B}^8)^{32})^* \times (\mathbb{B}^8)^*$.

Definition 7. *The execution environment, $\iota = (actor, input, sender, value, code)$, specifies the account responsible for the transaction, $sender$, the one currently executing, $actor$, the instructions executing under a given message, respectively $code$ and $input$, and the value to be transferred, $value$. The set of execution environments is denoted by $I = \mathbb{A} \times (\mathbb{B}^8)^* \times \mathbb{A} \times \mathbb{N}_{256} \times (\mathbb{B}^8)^*$.*

The elements specifying the information provided at the beginning of a transaction and the effects to apply afterwards, account destruction and refunding, are represented by the transaction environment and transaction effects, respectively.

Definition 8. *The transaction environment consists of a tuple - $(o, price, B)$ - containing immutable information relative to the transaction: the address of the account which originated the transaction - o,* *the gas price of the transaction - $price$, and the block - B, where the transaction is inserted. The set of transaction environments is denoted by $\mathcal{T}_{env} = \mathbb{A} \times \mathbb{N}_{256} \times \mathcal{B}$.*

Definition 9. *The transaction effect, $\eta = (bal_r, L, S_\dagger)$, contains the value to transfer to the beneficiary's account, bal_r, the sequence of log entries, $L \in \mathbb{A} \times (\{\} \cup (\mathbb{B}^{32}) \cup (\mathbb{B}^{32})^2 \cup (\mathbb{B}^{32})^3 \cup (\mathbb{B}^{32})^4) \times (\mathbb{B}^8)^*$, and contracts which were destroyed $S_\dagger \subseteq \mathbb{A}$.*

In this fashion, we denote the transaction effects set by N.

In conclusion, we can describe an execution state the top element of a call stack $(\mu, \iota, \sigma, \eta) :: S \in \mathbb{S}$. The call stack is limited to the depth of 1024 elements (equal to the limit of calls performed in a transaction specified in [44]).

Definition 10. *A call stack S consists of a stack with a depth of 1024 elements, which fall into four different categories, a plain call - $(\mu, \iota, \sigma, \eta)$, a halting execution state - $HALT(\sigma, g, d, \eta)$, reverting state - $REV(g, d)$ and an exception state - EXC (g and d representing the remaining gas after the execution and output data respectively). It obeys the last in first out principle, with the addition of an element being triggered by the start of an inner execution and the removal of the top execution taking place after a halting or exception occurrence. The current active execution state consists of the element on the top of the call stack.*

The set of the call stacks is denoted by $\mathbb{S} = M \times I \times \Sigma \times N$.

$$\begin{gathered} S = HALT(\sigma, g, d, \eta) :: S_{plain} | REV(g, d) :: S_{plain} | EXC :: S_{plain} | S_{plain} \\ S_{plain} = (\mu, \iota, \sigma, \eta) :: S_{plain} | (\mu, \iota, \sigma, \eta) \\ \mu = (gas, pc, m, i, s, rd) \\ \iota = (actor, input, sender, value, code) \\ \eta = (bal_r, L, S_\dagger) \end{gathered} \tag{3.1}$$

Equation 3.1 Context-free grammar for the call stack [23]

In addition to the only active execution state being the top element of the call stack, the information conveyed by this state to the next execution state is only transmitted when a halting or reverting configuration occurs. By opposition, the exceptional states do not carry any further information, as all the effects of the respective internal transaction are reverted.

The execution of each internal transaction starts in a fresh machine state, with an empty stack, memory initialized with all entries to zero, and program counter and active words in memory are set to zero. Only the gas is instantiated with the gas value available for the execution.

3.1.2 Small-step Relation

Before introducing the small-step relation, which describes the computations performed by the EVM stepwise, an auxiliary function is proposed by Grishchenko *et al.* [23] with the purpose of identifying the current opcode in execution:

$$\omega_{\mu,\iota} = \begin{cases} \iota.\mathsf{code}[\mu.\mathsf{pc}] & \mu.\mathsf{pc} < |\iota.\mathsf{code}| \\ \mathsf{STOP} & otherwise \end{cases} \tag{3.2}$$

Equation 3.2: Current instruction executing retrieval function

Bearing this in mind, the semantics of the EVM bytecode are described by the small-step relation, which is subject to a set of rules.

$$\frac{\omega_{\mu,\iota} = \mathsf{OP} \quad premises(S)}{\Gamma \vDash S \rightarrow S'} \qquad S, S' \in \mathbb{S},\ \ \Gamma \in \mathbb{T}_{env} \tag{3.3}$$

Equation 3.3: Small-Step Rule Structure

An important feature concerning the small-step relation $\rightarrow$ is that it is deterministic, i.e., for a given execution state $(\mu, \iota, \sigma, \eta) :: S \in \mathbb{S}$, under a transaction environment $\Gamma \in \mathbb{T}_{env}$, there can be at most one valid transition $\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \rightarrow S'$, for $S' \in \mathbb{S}$.

The beginning and completion of transactions is abstracted by two auxiliary functions: $initialize(\cdot,\cdot,\cdot) \in \mathbb{T} \times \mathcal{B} \times \Sigma \rightarrow (\mathbb{T}_{env} \times \mathbb{S}) \cup \{\bot\}$ and $finalize(\cdot,\cdot,\cdot) \in \mathbb{T} \times N \times \mathbb{S} \times \Sigma$. Furthermore, the formal representation of the change on the state σ operated by a transaction T is as given:

$$\frac{\begin{array}{c}(\Gamma, S) = initialize(T, B, \sigma) \\ \Gamma \vDash S \rightarrow^{*} S' \quad final(S') \quad \sigma' = finalize(T, \eta', S')\end{array}}{\sigma \xrightarrow{T,B} \sigma'} \tag{3.4}$$

Equation 3.4 Transaction formal display as in [23]

3.1.3 Operational Semantics

Structural operational semantics provide a one-step based means to reason with computations. Thus, as a matter of convenience and because we shall duel with the transitions regarding sequences of actions from a given execution state, we shall properly introduce the definition of the corresponding operational semantics.

An action is given by an opcode and the respective sequence of arguments (for the case of the PUSH instructions). The set of actions is denoted by Act.

The small-step relation portrays transitions between execution states corresponding to call stacks, $\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \rightarrow S'$, for a transaction environment $\Gamma \in \mathbb{T}_{env}$ and call stacks $(\mu, \iota, \sigma, \eta) :: S, S' \in \mathbb{S}$.

This transition is performed by the application of the rule relative to the action corresponding to the opcode on the program counter position of the executing code.

For the case of the evaluation of action sequences $\pi \in Act^*$, a transition is represented as $\Gamma \vDash S \xrightarrow{\pi}^* S' \rangle$ and obtained by successive applications of the corresponding small-step transition rules of each action in π.

Definition 11. *The relation $\to^*$ is obtained from the small-step relation $\to$ through addition of the two following rules:*

Given a transaction environment $\Gamma \in \mathbb{T}_{env}$ and call stacks S, S' and $S'' \in \mathbb{S}$

$$\frac{}{\Gamma \vDash S \to^* S} \qquad \frac{\Gamma \vDash S \to^* S' \quad \Gamma \vDash S' \to S''}{\Gamma \vDash S \to^* S''} \tag{3.5}$$

Equation 3.5 Evaluation Relation Rules

3.2 Optimization of EVM Bytecode Patterns

In order to reason with the execution of action sequences, it is necessary to introduce the notion of context adapted to the EVM semantics.

Definition 12. *An action sequence execution context denoted by $\mathcal{C}[\cdot]$ is described as an incomplete execution state given by a call stack - $(\mu, \iota, \sigma, \eta) :: S,\ \mu \in M,\ \iota \in I,\ \sigma \in \Sigma,\ \eta \in N,\ S \in \mathbb{S}$ - where $\sigma(a).\mathsf{code} = \iota.\mathsf{code}$ consists of $\alpha + +x + +\beta$, with $\alpha, \beta \in Act^*$, $|\alpha| = \mu.\mathsf{pc} - 1$, $a = \iota.\mathsf{actor}$ and $x \in X$ $(X$ a countable set of variables).*

An instance associated with a given action sequence $\pi \in Act^*$, $\mathcal{C}[\pi]$, consists of the substitution of the variable x for π.

The semantic equivalence between action sequences may be informally defined as:

Definition 13. *Two action sequences π and π' are semantically equivalent, denoted by $\pi \approx_{sem} \pi'$, if either can be replaced by the other in any program context conducing to the same final execution state.*

Due to the fact that our focus resides in optimizing gas consumption of the execution of Ethereum contracts, besides this notion of equivalence, a relation of optimization ought to be defined bearing in mind restrictions concerning gas availability of the final state and program counter.

Therefore, the optimization relation regarding EVM action sequences may be given as described below:

Definition 14. *For two action sequences $\{\pi, \pi'\}$, π' constitutes an optimization of π, $\pi' \prec_{opt} \pi$, if and only if for all transaction environments $\Gamma \in \mathbb{T}_{env}$, contexts $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$ and execution states $(\mu', \iota', \sigma', \eta') :: S'$, $HALT(\sigma'', g, d, \eta') :: S', HALT(\sigma'', g', d', \eta') :: S'$, $HALT(\sigma'', g, \varepsilon, \eta') :: S'$, $HALT(\sigma'', g', \varepsilon, \eta') :: S'$, $REV(g, d) :: S'$, $REV(g', d') :: S'$, $(\mu'', \iota'', \sigma'', \eta'') :: S' \in \mathbb{S}$, the given implications follow:*

- $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^* (\mu', \iota', \sigma', \eta') :: S' \implies \Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^* (\mu'', \iota'', \sigma'', \eta'') :: S'$
- $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^* HALT(\sigma', g, d, \eta') :: S' \implies \Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^* HALT(\sigma'', g', d', \eta') :: S'$
- $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^* HALT(\sigma', g, \varepsilon, \eta') :: S' \implies \Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^* HALT(\sigma'', g', \varepsilon, \eta') :: S'$
- $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^* REV(g, d) :: S' \implies \Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^* REV(g', d') :: S'$,

with $\mu'' = \mu'$, $\iota'' = \iota'$, $\sigma'' = \sigma'$, $\eta'' = \eta'$, $g' > g$, and $d' = d$, obeying the following additional conditions: $\mu''.\text{gas} > \mu'.\text{gas}$, $\iota'.\text{code} \neq \iota''.\text{code}$, $\sigma'(a).\text{code} \neq \sigma''(a).\text{code}$ (a standing for the contract address $\iota.\text{actor}$), and $\mu'.\text{pc}$ may differ from $\mu''.\text{pc}$.

Additionally, the actions to be optimized, must not produce an exceptional state for all contexts, i.e.,

$$\exists \Gamma \in \mathbb{T}_{env},\ \mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S,\ \textit{and}\ S' \in \mathbb{S} :\ \forall S'' \in \mathbb{S},\ \Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^* S',$$

with $S' \neq EXC :: S''$

Proposition 1. *The relation $\prec_{opt}$ is a strict partial order over Act^* and it is well-founded.*

Proof. By the definition strict partial order, it lacks to prove that $\prec_{opt}$ is an irreflexive and transitive relation contained in $Act^* \times Act^*$.

- For the first property, the irreflexiveness of $\prec_{opt}$ shall be proven by simply observing that the EVM is deterministic, which means that, given the action sequence $\pi \in Act^*$, transaction environment $\Gamma \in \mathbb{T}_{env}$ and context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$ there is only one possible resulting state, $S' \in \mathbb{S}$, from the transition $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^* S'$.

  As a consequence, if S' is either a plain execution state $(\mu, \iota, \sigma, \eta) :: S''$, an halting state $HALT(\sigma, g, d, \eta) :: S''$, or reverting state $REV(g, d) :: S''$, it is impossible to observe $\mu.\text{gas} > \mu.\text{gas}$ and $g > g$.

- The transitivity of $\prec_{opt}$ is simple to observe, as given the action sequences $\pi, \pi', \pi'' \in Act^*$, if $\pi \prec_{opt} \pi'$ and $\pi' \prec_{opt} \pi''$, then, by definition of $\prec_{opt}$, there exists a $\Gamma \in \mathbb{T}_{env}$ $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, such that $\Gamma \vDash \mathcal{C}[\pi''] \xrightarrow{\pi''}^* S''$, with S'' belonging to one of the four types below:

$$(\mu', \iota', \sigma', \eta') :: S', \quad HALT(\sigma', g, d, \eta) :: S', \quad HALT(\sigma', g, \varepsilon, \eta) :: S', \quad REV(g, d) :: S'$$

  Additionally, from the definition of the optimization relation, it must also hold that $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^* S'''$, with the execution state given by the call stack $S''' \in \mathbb{S}$ belonging in the same category as S''.
  A further application of the same argument leads us to conclude that there must also exist a $S'''' \in \mathbb{S}$, such that $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^* S''''$, with the execution state S'''' belonging in the same category as S''.
  Since $\pi' \prec_{opt} \pi''$ and $\pi \prec_{opt} \pi'$ both hold, then the returning data, memory, stack, storage and transaction effects are identical for the resulting states of the execution of the three sequences and we observe:

- $\mu'''.gas > \mu''.gas > \mu'.gas$ (for the case the three resulting states are plain states and μ''', μ'', μ' are the final machine states for π, π', π'', respectively).
- $g'' > g' > g$ (for the case the three resulting states are reverting or halting states, g'', g', g and are the remaining gas and returning data after the execution π, π', π'', respectively).

Additionally, since we have three different values for gas consumption, the action sequences (and the respective ι.code and $\sigma(a).code$, with a the address of ι.actor) are all different among them, thus allowing to conclude that $\pi \prec_{opt} \pi''$.

In order to prove that $\prec_{opt}$ is well-founded, it is necessary to state that every subset of action sequences has a least a minimal element with respect to $\prec_{opt}$, i.e., for $A \subseteq Act^*$, $\exists \pi \in A : \forall \pi' \in A\ \pi' \not\prec_{opt} \pi$.

Given a subset $A \subseteq Act^*$:

- There is an element i such that $\pi_i = \varepsilon$ - in this case, no action sequence leads to the initial configuration without the consumption of gas, thus, as $\nexists\, \pi \in Act^* :\ \pi \prec_{opt} \varepsilon$, the result holds.
- For every action $\pi \in A$, $\pi \neq \varepsilon$. In this case, either π is already a minimum element of A or there is an action sequence $\pi' \in A$ such that:

$$\pi' \prec_{opt} \pi \quad (\dagger)$$

. From $(\dagger)$ and from the definition of $\prec_{opt}$ it follows that:

$$\exists \Gamma \in \mathbb{T}_{env}, \mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S, S' \in \mathbb{S} :\ \Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^{*} S'',$$

such that S'' is of the form of one of the four execution states:

$$(\mu', \iota', \sigma', \eta') :: S', \quad HALT(\sigma', g, d, \eta) :: S', \quad HALT(\sigma', g, \varepsilon, \eta) :: S', \quad REV(g, d) :: S'$$

Additionally, also from the definition of the optimization relation and from $(\dagger)$, we have that:

$$\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^{*} S''',$$

with $S''' \in \mathbb{S}$ of the same type as S'', although, among other differences the parameter g' or μ''.gas of S'' is the condition that $g' > g$ or $\mu''.gas > \mu'.gas$ (depending on the type of execution state that both S'' and S''' exhibit).

Since $g,\ \mu$.gas $\in \mathbb{N}_{256}$, the initial available gas μ.gas is finite and so a limit to the optimization is set, thus deeming the possible optimizations we are able to perform to a finite number (ensuring the existence of a action sequence which does not allow further optimize, the minimum of $\prec_{opt}$).

$\square$

All the action sequences intended for optimization contain exclusively EVM instructions that operate solely on the execution stack, altering the program counter and available gas. Furthermore, in every pattern it is trivial to see that the optimized sequence requires less gas, which means that no gas exception will occur during the execution of the optimized sequence whilst the pattern to be optimized is evaluated successfully.

When referring to requirements for the successful execution, we allude to the premises of the rules present in the work of Xavier [45].

3.2.1 Patterns in the Literature

Proposition 2.

$$\varepsilon \quad \prec_{opt} \quad \begin{array}{l} \mathsf{SWAP}n \\ \mathsf{SWAP}n \end{array}$$

$n = 1, \dots, 16$

Proof. Since ε denotes the empty sequence and the transition described by $\Gamma \vDash \mathcal{C}[\varepsilon] \rightarrow^* \mathcal{C}[\varepsilon]$ is always valid, for $\Gamma \in \mathbb{T}_{env}$ and $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, it is left to prove that for $\pi = \mathsf{SWAP}n\ \mathsf{SWAP}n,\ n = 1 \dots 16$, $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^* (\mu', \iota', \sigma, \eta) :: S$, with $(\mu', \iota', \sigma, \eta) :: S \in \mathbb{S}$, only alters the $\mu.\mathsf{gas}$ and $\mu.\mathsf{pc}$.

By the rule for successful execution of SWAPn, the consecutive execution of the same instruction SWAPn leads to reverting the changes made to the stack (the top element returns to its prior position, as well as the $n + 1^{th}$ element).

$$\mu'.\mathsf{s} = s_0 :: \dots :: s_n :: s = \mu.\mathsf{s}$$

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 6$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 2$$

□

Proposition 3.

$$\mathsf{JUMPDEST} \quad \prec_{opt} \quad n \begin{cases} \mathsf{JUMPDEST} \\ \vdots \\ \mathsf{JUMPDEST} \end{cases}$$

$n \geq 2$

Proof. The rule for successful execution of JUMPDEST given the execution state $(\mu, \iota, \sigma, \eta) :: S \in \mathbb{S}$ only alters the $\mu.\mathsf{gas}$ and $\mu.\mathsf{pc}$ (decreasing the first by one unit of gas and increasing the second by one). □

Proposition 4.

$$\mathsf{POP} \quad \prec_{opt} \quad \begin{array}{l} \mathsf{OP} \\ \mathsf{POP} \end{array}$$

OP $\in$ {ISZERO, NOT, BALANCE, CALLDATALOAD, EXTCODESIZE,
BLOCKHASH, MLOAD, SLOAD}

Proof. All of the opcodes in the domain of OP share a common feature, they remove an element from the stack and place another on the top. As a consequence, if the execution of $\pi =$ OP POP succeeds, given the context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$ and transaction environment $\Gamma \in \mathbb{T}_{env}$, the element that is the result of the action OP placed on top of the stack is removed by the POP instruction.

Therefore, an equivalent and cheaper (sparing the gas consumption of OP) action sequence would consist of $\pi' =$ POP, which would remove from stack the element that is argument of OP instruction.

Let $(\mu', \iota', \sigma, \eta) :: S, (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$, the transitions

$$\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}{}^{*} (\mu', \iota', \sigma, \eta) :: S$$

$$\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}{}^{*} (\mu'', \iota'', \sigma, \eta) :: S$$

exhibit the following results:

$$\mu'.\text{s} = s_1 :: s = \mu''.\text{s},$$

with $\mu.\text{s} = s_0 :: s_1 :: s$

$$\mu'.\text{gas} \leq \mu.\text{gas} - 5$$

$$\mu''.\text{gas} = \mu.\text{gas} - 2$$

$$\mu'.\text{pc} = \mu.\text{pc} + 2$$

$$\mu''.\text{pc} = \mu.\text{pc} + 1$$

□

Proposition 5.

$$\begin{array}{c} \text{POP} \\ \text{POP} \end{array} \prec_{opt} \begin{array}{c} \text{OP} \\ \text{POP} \end{array}$$

OP $\in$ {ADD, SUB, MUL, DIV, SDIV, MOD, SMOD, EXP,
SIGEXTND, LT, GT, SLT, SGT, EQ, AND, OR, XOR, BYTE, SHA3}

Proof. This proof is identical to the one for the previous pattern, with the slight difference that all the opcodes in the specified domain for this pattern take two words from the stack instead of one. Therefore, a cheaper sequence alternative to $\pi =$ OP POP consists of removing the two arguments of OP from the stack, i.e., $\pi' =$ POP POP (spares the gas consumption of OP subtracted two gas units for the extra POP). It should be noticed that the requirements for the successful execution of both sequences are identical: excluding gas requirements, the stack must contain at least two elements.

Assuming the successful execution of $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}{}^{*} (\mu', \iota', \sigma, \eta) :: S$, which, as previously observed, implies the successful execution of $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}{}^{*} (\mu'', \iota'', \sigma, \eta) :: S$, given the transaction environment

$\Gamma \in \mathbb{T}_{env}$, context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$ and execution states $(\mu', \iota', \sigma, \eta) :: S,\ (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$.

$$\mu'.\mathsf{s} = s_2 :: s = \mu''.\mathsf{s},$$

with $\mu.\mathsf{s} = s_0 :: s_1 :: s_2 :: s$

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 2 - \texttt{opcost},$$

with $\texttt{opcost} \geq 3$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 4$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 2 = \mu''.\mathsf{pc}$$

□

Proposition 6.

$$\begin{array}{l} \mathsf{POP} \\ \mathsf{POP} \\ \mathsf{POP} \end{array} \prec_{opt} \begin{array}{l} \mathsf{OP} \\ \mathsf{POP} \end{array}$$

$\mathsf{OP} \in \{\mathsf{ADDMOD}, \mathsf{MULMOD}\}$

Proof. Similarly to what was stated for the previous pattern, the proof shall take into account that all the opcodes in the specified domain for OP take three arguments instead of one. Therefore, a cheaper sequence alternative to $\pi =$ OP POP consists of popping the two arguments of OP from the stack, i.e., $\pi' =$ POP POP POP (spares the gas consumption of OP subtracted four gas units, concerning the extra two POP instructions). It should be noticed that the requirements for the successful execution of both sequences are identical, excluding gas requirements, the stack must contain at least three elements.

Assuming the successful execution of $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^{*} (\mu', \iota', \sigma, \eta) :: S$, which, as previously observed, implies $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^{*} (\mu'', \iota'', \sigma, \eta) :: S$, given the transaction environment $\Gamma \in \mathbb{T}_{env}$, context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$ and execution states $(\mu', \iota', \sigma, \eta) :: S,\ (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$.

$$\mu'.\mathsf{s} = s_3 :: s = \mu''.\mathsf{s},$$

with $\mu.\mathsf{s} = s_0 :: s_1 :: s_2 :: s_3 :: s$

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 2 - 8$$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 6$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 2$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{pc} + 3$$

□

Proposition 7.

$$\varepsilon \quad \prec_{opt} \quad \begin{array}{c}\mathsf{OP}\\ \mathsf{POP}\end{array}$$

$\mathsf{OP} \in \{\mathsf{ADDRESS}, \mathsf{ORIGIN}, \mathsf{CALLER}, \mathsf{CALLVALUE}, \mathsf{CALLDATASIZE}, \mathsf{CODESIZE}, \mathsf{GASPRICE}, \mathsf{COINBASE}, \mathsf{TIMESTAMP}, \mathsf{NUMBER}, \mathsf{DIFFICULTY}, \mathsf{GASLIMIT}, \mathsf{PC}, \mathsf{MSIZE}, \mathsf{GAS}\}$

Proof. As given the transaction environment $\Gamma \in \mathbb{T}_{env}$ and context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, the transition $\Gamma \vDash \mathcal{C}[\varepsilon] \to^* \mathcal{C}[\varepsilon]$ is always valid, it suffices only to prove that the successful execution of $\pi = \mathsf{OP}\ \mathsf{POP}$, represented by $\Gamma \vDash \mathcal{C}[\pi] \to^* (\mu', \iota', \sigma, \eta) :: S$, with $(\mu', \iota', \sigma, \eta) :: S \in \mathbb{S}$.

All the actions that OP may represent only place an element on the stack, which is popped through the execution of the following instruction, POP. Therefore, all the changes to the stack are reverted and only gas is consumed (an amount greater or equal 4 units) and the program counter increased (by two) when executing π.

$$\mu'.\mathsf{s} = \mu.\mathsf{s}$$

$$\mu'.\mathsf{gas} \leq \mu.\mathsf{s} - 4$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 2$$

□

Proposition 8.

$$\begin{array}{c}\mathsf{DUP2}\\ \mathsf{SWAP}n+1\\ \mathsf{OP}'\\ \mathsf{OP}\end{array} \quad \prec_{opt} \quad \begin{array}{c}\mathsf{SWAP1}\\ \mathsf{SWAP}n\\ \mathsf{OP}'\\ \mathsf{DUP}n\\ \mathsf{OP}\end{array}$$

$n = 2, \ldots, 15; \quad \mathsf{OP}, \mathsf{OP}' \in \{\mathsf{ADD}, \mathsf{MUL}, \mathsf{AND}, \mathsf{OR}, \mathsf{XOR}\}$

Proof. Executing consecutively $\mathsf{SWAP1}\ \mathsf{SWAP}n$ or $\mathsf{DUP2}\ \mathsf{SWAP}n+1$ for the circumstances described ($n \geq 2$), both solely require the context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$ to be such that $|\mu.\mathsf{s}| \geq n+1$. Additionally, both $\mathsf{DUP2}$ and $\mathsf{DUP}n$ are responsible for placing an element on top of the execution stack, thus ensuring that if OP doesn't occur in the optimized sequence, then its execution in the non optimized sequence would also fail.

As a consequence, the successful execution of $\pi = \mathsf{SWAP1}\ \mathsf{SWAP}n\ \mathsf{OP}'\ \mathsf{DUP}n\ \mathsf{OP}$ entails the one of $\pi' = \mathsf{DUP2}\ \mathsf{SWAP}n+1\ \mathsf{OP}'\ \mathsf{OP}$.

Hence, given the transaction environment $\Gamma \in \mathbb{T}_{env}$ and execution states $(\mu', \iota', \sigma, \eta) :: S$, $(\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$, the transitions $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}{}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}{}^{*} (\mu'', \iota'', \sigma, \eta) :: S$

exhibit the following characteristics:

$$\mu''.\mathsf{s} = \mathsf{op}'(s_n, s_0) :: \mathsf{op}(s_1, s_2) :: \ldots :: s_{n-1} :: s_1 :: s \ = \mu'.\mathsf{s},$$

with $\mu.\mathsf{s} = s_0 :: \ldots :: s_{n-1} :: s_n :: s$.

$$\mu''.\mathsf{gas} = \mu'.\mathsf{gas} + 3$$

$$\mu''.\mathsf{pc} = \mu'.\mathsf{pc} - 1$$

□

Proposition 9.

$$\mathsf{STOP} \quad \prec_{opt} \quad \begin{array}{c}\mathsf{OP}\\ \mathsf{STOP}\end{array}$$

OP *can be any operation except* JUMPDEST*,* JUMP*,* JUMPI *and all operations that change storage.*

Proof. This pattern's optimization is straightforward, as any action that only modifies memory or the execution stack (structures bounded to a single message call) followed by the STOP operation and consequent execution halting would not produce any change in the global state.

Therefore, given the transaction environment $\Gamma \in \mathbb{T}_{env}$ and context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$ executing $\pi =$ OP STOP and $\pi' =$ STOP leads to the following transitions:

$$\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}{}^{*} HALT(\sigma, g, d, \eta) :: S$$

$$\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}{}^{*} HALT(\sigma, g', d, \eta) :: S,$$

with $g' = g + \texttt{opcost}$, where opcost stands for the cost of OP. □

Proposition 10.

$$\begin{array}{c}\mathsf{DUP1}\\ \mathsf{SWAP}n+1\end{array} \quad \prec_{opt} \quad \begin{array}{c}\mathsf{SWAP}n\\ \mathsf{DUP}n+1\\ \mathsf{SWAP1}\end{array}$$

$n = 1, \ldots, 15$

Proof. In order to successfully execute the action sequence $\pi =$ SWAPn DUP$n+1$ SWAP1, the context given, $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, must be so that $|\mu.\mathsf{s}| \geq n+1$, which is a strong enough condition, allied to the assumption that the gas available is sufficient, for $\pi' =$ DUP1 SWAP$n+1$ to also evaluate to a plain execution state.

Taking in consideration a transaction environment $\Gamma \in \mathbb{T}_{env}$, the transitions $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}{}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi}{}' (\mu'', \iota'', \sigma, \eta) :: S$, for $(\mu', \iota', \sigma, \eta) :: S$,

$(\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$, lead to:

$$\mu''.\mathsf{s} = s_n :: s_0 :: \ldots :: s_{n-1} :: s_0 :: s = \mu'.\mathsf{s}$$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 6$$

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 9$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{pc} + 2$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 3,$$

with $\mu.\mathsf{s} = s_0 :: \ldots :: s_{n-1} :: s_n :: s$. □

Proposition 11.

$$\begin{array}{c} m \textit{ consecutive } \mathsf{POP} \\ \mathsf{PUSH}n \end{array} \prec_{opt} \begin{array}{r} \mathsf{PUSH}n \\ \mathsf{SWAP}m \\ m \textit{ consecutive } \mathsf{POP} \end{array}$$

$n = 1, \ldots, 32; \quad m = 1, \ldots, 16$

Proof. If the execution $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}{}^{*} (\mu', \iota', \sigma, \eta) :: S$ occurs with success, for the action sequence $\pi = \mathsf{PUSH}n\ \mathsf{SWAP}m\ m$ consecutive POP, context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, and execution state $(\mu', \iota', \sigma, \eta) :: S \in \mathbb{S}$, it comes as a consequence from the application of the rules for successful PUSHn and SWAPm that the stack would have at least m elements at the beginning of the execution of this sequence, i.e., $m \leq |\mu.\mathsf{s}| < 32$. Therefore, the only preconditions for executing m consecutive POP are met and given this sequence, evidently, the stack is not at full capacity after the execution, thus permitting the PUSHn action.

The resulting state of $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}{}^{*} (\mu'', \iota'', \sigma, \eta) :: S$, for the action sequence $\pi' = m$ consecutive POP PUSHn, execution state $(\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$ is such that:

$$\mu''.\mathsf{s} = a :: s = \mu'.\mathsf{s}$$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 3 - 2 * m$$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 3 - 2 * m - 3$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{pc} + m + 1$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + m + 2,$$

with $\mu.\mathsf{s} = s_0 :: \ldots :: s_{m-1} :: s$ and a the argument of PUSHn. □

Proposition 12.

$$n+1 \textit{ consecutive } \mathsf{POP} \quad \prec_{opt} \quad \begin{array}{r}\mathsf{SWAP}n \\ n+1 \textit{ consecutive } \mathsf{POP}\end{array}$$

$n = 1, \ldots, 16$

Proof. The successful execution of $\pi = \mathsf{SWAP}n\ \ n+1$ consecutive POP, given the context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$ and transaction environment $\Gamma \in \mathbb{T}_{env}$ is represented by $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^{*} (\mu', \iota', \sigma, \eta) :: S$, for $(\mu', \iota', \sigma, \eta) :: S \in \mathbb{S}$, proceeds in a way such that:

$$\mu'.\mathsf{s} = s_{n+1} :: s,$$

with $\mu.\mathsf{s} = s_0 :: \ldots :: s_{n+1} :: s$

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 2 * (n+1) - 3$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + (n+1) + 1$$

This execution state is identical to the one obtained by executing $\pi = n+1$ consecutive POP from the same context $\mathcal{C}[\cdot]$. Furthermore, the SWAPn is superfluous due to the following $n+1$ POP instructions, as either the first and the $n+1^{th}$ $\mu.\mathsf{s}$ stack elements are removed. Moreover, as SWAP instructions do not place additional elements in the stack, the requirements for successfully executing SWAPn and $n+1$ consecutive POP are the same (the stack of the initial execution state must contain at least $n+1$ elements).

Therefore, executing $\pi' = n+1$ consecutive POP within the same transaction environment Γ and context $\mathcal{C}[\cdot]$ and given that π executes successfully, the execution state $(\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$ is achieved.

$$\mu''.\mathsf{s} = s_{n+1} :: s = \mu'.\mathsf{s}$$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 2 * (n+1) = \mu'.\mathsf{gas} + 3$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{pc} + (n+1) = \mu'.\mathsf{pc} + 1$$

□

Proposition 13.

$$\begin{array}{r} n \textit{ consecutive } \mathsf{POP} \\ \mathsf{SWAP}m-n \\ m-n \textit{ consecutive } \mathsf{POP}\end{array} \quad \prec_{opt} \quad \begin{array}{r}\mathsf{SWAP}n \\ \mathsf{SWAP}m \\ m \textit{ consecutive } \mathsf{POP}\end{array}$$

$n, m = 1, \ldots, 15; \quad n < m$

Proof. Executing a SWAPn instruction followed by a SWAPm and a sequence of m POP instructions, i.e., a sequence $\pi =$ SWAPn SWAPm m consecutive POP, with $n < m$ deems the first SWAPn irrelevant, as either the top and the $n+1^{th}$ element of the stack are popped during the execution of POP instructions. Therefore a more efficient sequence is obtained by $\pi' = n$ consecutive POP SWAP$m-n$ $m-n$ consecutive POP and since one condition for the execution of π is for the stack to have $m+1$ elements prior to execution of the action sequence, the requirements for π' to evaluate to a non exceptional state are fulfilled.

Accordingly the changes in the execution state induced by $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^{*} (\mu'', \iota'', \sigma, \eta) :: S$, for $\Gamma \in \mathbb{T}_{env}$, $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$ and $(\mu', \iota', \sigma, \eta) :: S$, $(\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$, lead to:

$$\mu'.\mathsf{s} = s_{n+1} :: s = \mu''.\mathsf{s},$$

with $\mu.\mathsf{s} = s_0 :: \ldots :: s_n :: s_{n+1} :: s$

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 6 - 2 \times m$$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 3 - 2 \times m$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 2 + m$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{pc} + 1 + m$$

□

Proposition 14.

$$\begin{array}{ll} x-y \textit{ consecutive } \mathsf{PUSH}n, & \textit{if } x>y; \\ y-x \textit{ consecutive } \mathsf{POP}, & otherwise \end{array} \prec_{opt} \begin{array}{c} x \textit{ consecutive } \mathsf{PUSH}n \\ y \textit{ consecutive } \mathsf{POP} \end{array}$$

$n = 1, \ldots, 32$

Proof. A sequence of POP instructions following actions such as PUSHn, which places an element on top of the stack, reverts the changes that PUSHn operates on the stack, implying a decrease in terms of gas and program counter only.

If the PUSH sequence is longer than the POP sequence, the exceeding PUSH instructions (starting at the bottom, the PUSH instructions after the y^{th} PUSH) still operate changes in the execution stack and shall be accounted for an optimization. Otherwise, the remaining POP also operates changes on the stack and must remain in the optimized sequence.

The gas saved by this pattern replacement is y, if the number of POP instructions exceeds the number of PUSH, and $3 * x$, for the reverse case. □

Proposition 15.

$$\begin{array}{ll} x-y \textit{ consecutive } \mathsf{DUP}n, & \textit{if } x>y; \\ y-x \textit{ consecutive } \mathsf{POP}, & otherwise \end{array} \prec_{opt} \begin{array}{c} x \textit{ consecutive } \mathsf{DUP}n \\ y \textit{ consecutive } \mathsf{POP} \end{array}$$

$n = 1, \ldots, 16$

Proof. A sequence of POP instructions following actions such as DUPn, which places a copy of the n^{th} element of the stack on the top, reverts the changes that DUPn operates on the stack, implying a decrease in terms of gas and program counter only.

If the sequence of DUP is longer than the POP sequence, the exceeding DUP instructions (starting at the bottom, the DUP instructions after the y^{th} DUP) still operate change in the execution stack and shall be included in an optimization. Otherwise, the remaining POP also operate changes on the stack and must remain in the optimized sequence.

The gas saved by this pattern replacement is y, if the number of POP instructions exceeds the number of DUP and $3 * x$ for the reverse case. □

Proposition 16.

$$\text{DUP}n \quad \prec_{opt} \quad \begin{array}{c}\text{DUP}n\\ \text{SWAP}n\end{array}$$

$n = 1, \ldots, 16$

Proof. Since the sequence $\pi' = \text{DUP}n$ corresponds to the initial action of $\pi = \text{DUP}n\ \text{SWAP}n$, the successful execution of π entails an identical result for the execution of π'. As SWAPn is responsible for the exchange between the top entry and the $n+1^{th}$, to execute it from a state where they are the exact copy of one another will only decrease the available gas.

As a consequence, given the transaction environment $\Gamma \in \mathbb{T}_{env}$ and context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, the evaluation of both sequences ($\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^{*} (\mu'', \iota'', \sigma, \eta) :: S$ with $(\mu', \iota', \sigma, \eta) :: S,\ (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$) produces the results given below:

$$\mu'.\text{s} = s_n :: s_0 :: \ldots :: s_n :: s = \mu''.\text{s}$$

with $\mu.\text{s} = s_0 :: \ldots :: s_n :: s$

$$\mu'.\text{gas} = \mu.\text{gas} - 6$$

$$\mu''.\text{gas} = \mu.\text{gas} - 3$$

$$\mu'.\text{pc} = \mu.\text{pc} + 2$$

$$\mu''.\text{pc} = \mu.\text{pc} + 1$$

□

Proposition 17.

$$\begin{array}{c} n-1 \textit{ consecutive } \mathsf{POP} \\ \mathsf{SWAP}1 \end{array} \prec_{opt} \begin{array}{r} \mathsf{SWAP}1 \\ \mathsf{SWAP}2 \\ \vdots \\ \mathsf{SWAP}n \\ \mathsf{SWAP}n-1 \\ \vdots \\ \mathsf{SWAP}1 \\ n-1 \textit{ consecutive } \mathsf{POP} \end{array}$$

$n = 2, \ldots, 16$

Proof. The sequence of consecutive SWAP actions requires the execution stack to have a depth of at least $n+1$ elements, thus, the successful execution of $\pi = \mathsf{SWAP}1\ \mathsf{SWAP}2\ \ldots\ \mathsf{SWAP}n\ \mathsf{SWAP}n-1\ \ldots\ \mathsf{SWAP}1\ n-1$ consecutive POP entails the successful execution of $\pi' = n-1$ consecutive POP SWAP1.

As a result of the execution of π and π' given the transaction environment $\Gamma \in \mathbb{T}_{env}$ and context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, two execution states, $(\mu', \iota', \sigma, \eta) :: S$ (from the transition described by $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^{*} (\mu', \iota', \sigma, \eta) :: S$) and $(\mu'', \iota'', \sigma, \eta) :: S$ (from the transition described by $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^{*} (\mu'', \iota'', \sigma, \eta) :: S$), are reached.

$$\mu''.\mathsf{s} = s_n :: s_{n-1} :: s = \mu'.\mathsf{s}$$

with $\mu.\mathsf{s} = s_0 :: \ldots :: s_{n-1} :: s_n :: s$.

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 2 \times (n-1) - 3$$

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 2 \times (n-1) - 3 * (2 * n - 1)$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{pc} + n$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 3 \times n - 2$$

□

Proposition 18.

$$\mathsf{OP} \prec_{opt} \begin{array}{c} \mathsf{SWAP}1 \\ \mathsf{OP} \end{array}$$

$\mathsf{OP} \in \{\mathsf{ADD}, \mathsf{MUL}, \mathsf{AND}, \mathsf{OR}, \mathsf{XOR}\}$

Proof. Operations such as addition and multiplication modulo 2^{256}, bitwise logic operations such as

$\{\wedge, \vee, \oplus\}$ are commutative, thus, as the instruction SWAP1 does not interfere with the requirements of the action OP, the successful execution of action sequence $\pi =$ SWAP1 OP entails the one of $\pi' =$ OP.

Given the transaction environment $\Gamma \in \mathbb{T}_{env}$ and context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, the evaluation of both sequences ($\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^{*} (\mu'', \iota'', \sigma, \eta) :: S$, with $(\mu', \iota', \sigma, \eta) :: S,\ (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$) produces the results below:

$$\mu'.\mathsf{s} = \mathtt{op}(s_0, s_1) :: s = \mu''.\mathsf{s},$$

with $\mu.\mathsf{s} = s_0 :: s_1 :: s$ and op the operation associated with the opcode OP.

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - \mathtt{opcost} - 3$$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - \mathtt{opcost},$$

with opcost representing the cost of executing the opcode OP.

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 2$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{pc} + 1$$

$\square$

Proposition 19.

$$\mathsf{OP} \quad \prec_{opt} \quad \begin{array}{c}\mathsf{OP}\\ \mathsf{ISZERO}\\ \mathsf{ISZERO}\end{array}$$

$\mathsf{OP} \in \{\mathsf{LT}, \mathsf{GT}, \mathsf{SLT}, \mathsf{SGT}, \mathsf{EQ}\}$

Proof. All the predicate operations that OP may take are binary operations that place on top of the stack either 0x01 or 0x00, representing the boolean values True and False that the expression of the predicate type operation applied to the first two stack items may take.

The opcode ISZERO is responsible for placing on the stack the value of the expression stating that the top element is equal to 0x00 (0x01, or 0x00 otherwise). Therefore, two consecutive ISZERO instructions applied to a boolean value, 0x01 or 0x00, corresponds to a double negation.

Given the transaction environment $\Gamma \in \mathbb{T}_{env}$, context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$ and action sequences $\pi =$ OP and $\pi' =$ OP, the evaluation of both sequences ($\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^{*} (\mu'', \iota'', \sigma, \eta) :: S$ with $(\mu', \iota', \sigma, \eta) :: S,\ (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$) produces the following result:

$$\mu'.\mathsf{s} = \mathtt{op}(s_0, s_1) :: s = \mu''.\mathsf{s},$$

with $\mu.\mathsf{s} = s_0 :: s_1 :: s$ and $\mathtt{op}$ the operation associated with the opcode OP.

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - \mathtt{opcost} - 6,$$

with $\mathtt{opcost}$ representing the cost of executing the opcode OP.

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - \mathtt{opcost}$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 3$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{pc} + 1$$

$\square$

Proposition 20.

$$\textit{combination of } n \ \mathsf{PUSH}x \quad \prec_{opt} \quad \begin{array}{r} n \textit{ consecutive } \mathsf{PUSH}x \\ \mathsf{SWAP}y \end{array}$$

$y < n; \quad x = 1, \ldots, 32; \quad y = 1, \ldots, 16$

Proof. This pattern consists of substituting a SWAP instruction when the contents of the stack this instruction operates on are directly specified in a sequence of PUSH preceding it.

Given that the $y \leq n$, then all the elements of the stack that SWAP will act upon correspond to arguments of the PUSH sequence. As a consequence, changing the order of the sequence suffices in order to produce the same effect on the stack, sparing the gas of the SWAP instruction.

It is trivial to see that the successful execution of $\pi = n$ consecutive PUSH SWAPy implies the same result for $\pi' =$ combination of n PUSH, as both first n instructions of both sequences consist of PUSH instructions, which for the case of π' corresponds to the whole sequence. $\square$

Proposition 21.

$$\begin{array}{r} \mathsf{PUSH}z \\ \mathsf{PUSH}x \\ \mathsf{SWAP}y+1 \end{array} \quad \prec_{opt} \quad \begin{array}{c} \mathsf{PUSH}x \\ \mathsf{SWAP}y \\ \mathsf{PUSH}z \\ \mathsf{SWAP}1 \end{array}$$

$x, z = 1, \ldots, 32; \quad y = 1, \ldots, 16$

Proof. The successful execution of the action sequence $\pi =$ PUSHx SWAPy PUSHz SWAP1 ensures that before the last command, SWAP1, is executed, the stack has at least $y + 2$ elements, as a SWAPy has successful been performed (the stack should have at least $y + 1$ elements by the rules of SWAP), followed by a PUSH, which is responsible for placing an additional item on top. Therefore, the successful execution of π entails an identical result for $\pi' =$ PUSHz PUSHx SWAP$y + 1$.

Given a transaction environment $\Gamma \in \mathbb{T}_{env}$, context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$ and execution states $(\mu', \iota', \sigma, \eta) :: S, (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$, the following executions of both action sequences π and π' - $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}{}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}{}^{*} (\mu'', \iota'', \sigma, \eta) :: S$ - are responsible for the transformations described below:

$$\mu'.\mathsf{s} = s_{y-1} :: a :: \ldots :: s_{y-2} :: b :: s = \mu''.\mathsf{s},$$

with $\mu.\mathsf{s} = s_0 :: \ldots :: s_{y-1} :: s$ and a the argument of PUSHz and b the argument of PUSHx.

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 12$$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 0$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 4 + z + x$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{pc} + 3 + z + x$$

□

Proposition 22.

$$\begin{array}{r} x \textit{ consecutive } \mathsf{PUSH}n \\ \mathsf{DUP}y \end{array} \approx_{sem} \textit{combination of } x+1 \ \mathsf{PUSH}n$$

$y \leq x; \quad n = 1, \ldots, 32; \quad y = 1, \ldots, 16$

Proof. This pattern is not an optimization of gas, consisting instead of, given a sequence more computationally complex $\pi = x$ consecutive PUSHn DUPy replacing a DUP instruction when the contents of the stack these instructions operate on are directly specified in a sequence of PUSH preceding it, $\pi' =$ combination of $x + 1$ PUSHn.

The opcode DUP does not change the execution stack other than placing an additional element on top of it. Nevertheless, in order to do so, it removes all the elements until it retrieves the y^{th} item, placing them back again afterwards.

As the element on which the DUPy instruction operates on is pushed to the stack during the execution of π and the execution cost of the opcodes DUP and PUSH is identical, the requirements for the successful execution of both π and π' are the same.

Given the transaction environment $\Gamma \in \mathbb{T}_{env}$ and context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, the evaluation of both sequences ($\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}{}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}{}^{*} (\mu'', \iota'', \sigma, \eta) :: S$, with $(\mu', \iota', \sigma, \eta) :: S,\ (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$) produces the following result:

$$\mu'.\mathsf{s} = p_{x-n} :: p_x :: \ldots :: p_0 :: s = \mu''.\mathsf{s},$$

where p_i represents the argument of the i^{th} PUSH and $\mu.\mathsf{s} = s$.

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 3 * (x + 1) = \mu''.\mathsf{gas}$$

□

Proposition 23.

$$\begin{array}{c} m \textit{ consecutive } \mathsf{OP} \\ n+m \textit{ consecutive and same } \mathsf{OP}' \end{array} \prec_{opt} \begin{array}{r} \mathsf{SWAP}n \\ m \textit{ consecutive } \mathsf{OP} \\ n+m \textit{ consecutive and same } \mathsf{OP}' \end{array}$$

OP $\in \{\mathsf{PUSH}x, \mathsf{DUP}y\}$; OP′ $\in \{\mathsf{ADD}, \mathsf{MUL}, \mathsf{AND}, \mathsf{OR}, \mathsf{XOR}\}$; $x = 1, \ldots, 32$; $n = 1, \ldots, 16$; $y = 1, \ldots, 16$

Proof. The principle underlying the optimization performed by this pattern substitution is equivalent to the one of Pattern 17. The fact that the operations in the domain of OP′ all enjoy commutativeness implies the rearrange for which the SWAPn instruction is responsible to be superfluous.

Therefore, executing π = SWAPn m consecutive OP $n+m$ consecutive and same OP′ and π' = m consecutive OP $n+m$ consecutive and same OP′, given the transaction environment $\Gamma \in \mathbb{T}_{env}$ and context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, the evaluation of both sequences ($\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^{*} (\mu'', \iota'', \sigma, \eta) :: S$, with $(\mu', \iota', \sigma, \eta) :: S,\ (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$) leads to the following result:

$$\mu'.\mathsf{s} = \mathsf{op}'(op_1, \ldots, op_m, s_n, s_1, \ldots, s_0) :: s = \mu''.\mathsf{s},$$

where $\mathsf{op}'(op_1, \ldots, op_m, s_n, s_1, \ldots, s_0)$ represents the application of OP′ to all the arguments pairwise and $\mu.\mathsf{s} = s_0 :: \ldots :: s_n :: s$.

$$\mu'.\mathsf{gas} = \mu''.\mathsf{gas} - 3$$

$$\mu'.\mathsf{pc} = \mu''.\mathsf{pc} + 1$$

□

Proposition 24.

$$\begin{array}{r} \mathsf{DUP}1 \\ \mathsf{DUP}2 \\ \mathsf{SWAP}n+1 \end{array} \prec_{opt} \begin{array}{r} \mathsf{DUP}1 \\ \mathsf{SWAP}n \\ \mathsf{DUP}2 \\ \mathsf{SWAP}1 \end{array}$$

$n = 1, \ldots, 15$

Proof. According the rules of DUP and SWAP, the successful execution $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^{*} (\mu', \iota', \sigma, \eta) :: S$, for a transaction environment $\Gamma \in \mathbb{T}_{env}$, action sequence

π = DUP1 SWAPn DUP2 SWAP1, context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$ and call stack $(\mu', \iota', \sigma, \eta) :: S \in \mathbb{S}$, implies that length of $\mu.\mathsf{s}$ is at least two elements shorter than the stack's maximum length, as no operation in π is responsible for removing items from the stack and the two DUP instructions place two items on top of it.

The sequence DUP1 SWAPn requires $|\mu.\mathsf{s}|\geq n+1$, which is sufficient for the successful execution $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}{}^{*} (\mu'', \iota'', \sigma, \eta) :: S$.

Therefore, all conditions for execution of $\pi' =$ DUP1 DUP2 SWAP$n+1$ are met and the resulting states of executing both π and π' from the context $\mathcal{C}[\cdot]$ are such that:

$$\mu'.\mathsf{s} = s_{n-1} :: s_0 :: \ldots :: s_0 :: s = \mu''.\mathsf{s},$$

for $\mu.\mathsf{s} = s_0 :: \ldots :: s_{n-1} :: s$

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 12$$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 9$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 4$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{pc} + 3$$

□

Proposition 25.

$$\begin{array}{c} \text{DUP}n \\ \text{DUP1} \\ \text{SWAP}n \\ \text{SWAP2} \end{array} \prec_{opt} \begin{array}{c} \text{DUP}n \\ \text{SWAP}n-1 \\ \text{SWAP1} \\ \text{DUP}n \\ \text{SWAP1} \end{array}$$

$n = 3, \ldots, 16$

Proof. As the operation DUPn occurs at the beginning of both $\pi =$ DUPn SWAP$n-1$ SWAP1 DUPn SWAP1 and $\pi' =$ DUPn DUP1 SWAPn SWAP2, for a successful execution of both sequences, it demands that the execution stack must have at least n elements at the start. With this statement, all of the requirements for π' to successfully execute are fulfilled, assuming the necessary gas is provided and given that the execution of π did not produce an exception.

For a transaction environment $\Gamma \in \mathbb{T}_{env}$, context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$ and call stacks $(\mu', \iota', \sigma, \eta) :: S$, $(\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$, the following executions of both action sequences π and π' - $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}{}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}{}^{*} (\mu'', \iota'', \sigma, \eta) :: S$ - are responsible for the transformations as stated below:

$$\mu'.\mathsf{s} = s_0 :: s_{n-1} :: s_{n-2} :: \ldots :: s_{n-3} :: s_{n-1} s_{n-1} :: s = \mu''.\mathsf{s},$$

for $\mu.\mathsf{s} = s_0 :: \ldots :: s_{n-2} :: s_{n-1} :: s$.

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 15$$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 12$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 5$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{pc} + 4$$

□

3.2.2 Patterns Discovered

Proposition 26.

$$\varepsilon \quad \prec_{opt} \quad \begin{array}{r} \mathsf{PUSH1}\ \mathtt{0x00} \\ \mathsf{OP} \end{array}$$

$\mathsf{OP} \in \{\mathsf{ADD}, \mathsf{SUB}\}$

Proof. Bearing in mind that the integer addition (subtraction) by zero (the neutral element of addition), i.e., evaluating the action sequence $\pi = \mathsf{PUSH1}\ \mathtt{0x00}\ \mathsf{ADD}$ ($\pi = \mathsf{PUSH1}\ \mathtt{0x00}\ \mathsf{SUB}$), given the transaction environment $\Gamma \in \mathbb{T}_{env}$ and context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, conduces either to an unaltered execution stack and less available gas or to an exceptional state. Therefore, in case of success, for the execution state $(\mu', \iota', \sigma, \eta) :: S \in \mathbb{S}$, $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}{}^{*} (\mu', \iota', \sigma, \eta) :: S$ is such that:

$$\mu'.\mathsf{s} = s_0 :: s = \mu.\mathsf{s},$$

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 6$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 2$$

□

Proposition 27.

$$\mathsf{PUSH20}\ \mathtt{0x}\underbrace{\mathtt{f}\ldots\mathtt{f}}_{40} \quad \prec_{opt} \quad \begin{array}{r} \mathsf{PUSH1}\ \mathtt{0x01} \\ \mathsf{PUSH1}\ \mathtt{0xa0} \\ \mathsf{PUSH1}\ \mathtt{0x02} \\ \mathsf{EXP} \\ \mathsf{SUB} \end{array}$$

Proof. The successful execution of $\pi = \mathsf{PUSH1}\ \mathtt{0x01}\ \mathsf{PUSH1}\ \mathtt{0xa0}\ \mathsf{PUSH1}\ \mathtt{0x02}\ \mathsf{EXP}\ \mathsf{SUB}$ obviously entails the one of the optimizing sequence, as its only action consists of a PUSH instruction, only differing in respect to the number of bytes pushed to the top of the execution stack and its argument to the first instruction of π.

Therefore, the successful execution of π entails also a favorable result to $\pi' = \mathsf{PUSH20}\ \mathtt{0x}\underbrace{\mathtt{f}\ldots\mathtt{f}}_{40}$.

Provided the transaction environment $\Gamma \in \mathbb{T}_{env}$ and context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, the transitions $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}{}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}{}^{*} (\mu'', \iota'', \sigma, \eta) :: S$, for $(\mu', \iota', \sigma, \eta) :: S,\ (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$, are such that:

$$\mu'.\text{s} = 0\text{x}\underbrace{\texttt{f}\ldots\texttt{f}}_{40} :: s = \mu''.\text{s},$$

with $\mu.\text{s} = s$.

$$\mu'.\text{gas} = \mu.\text{gas} - 3 * 3 - 10 - 3$$

$$\mu''.\text{gas} = \mu.\text{gas} - 3$$

$$\mu'.\text{pc} = \mu.\text{pc} + 8$$

$$\mu''.\text{pc} = \mu.\text{pc} + 21$$

□

Proposition 28.

$$\begin{array}{c} \text{PUSH}n\ \texttt{a} \\ \text{AND} \end{array} \prec_{opt} \begin{array}{c} \text{PUSH}n\ \texttt{a} \\ \text{AND} \\ \text{PUSH}n\ \texttt{a} \\ \text{AND} \end{array}$$

$n = 1, \ldots, 32$

Proof. Applying the bitwise logical conjunction to a given element, a, with another, b, twice, i.e., $(a \wedge b) \wedge b$, holds the same boolean value as $a \wedge b$, as can be seen by examining the truth table below:

a b	a ∧ b	(a ∧ b) ∧ b
1 1	1	1
1 0	0	0
0 1	0	0
0 0	0	0

Additionally, the sequence $\pi' = \text{PUSH}n\ \texttt{a}\ \ \text{AND}$ is contained in the original action sequence $\pi = \text{PUSH}n\ \texttt{a}\ \ \text{AND PUSH}n\ \texttt{a}\ \ \text{AND}$, thus the successful execution of of π entails the same result for the execution of π'. As a consequence, given the $\Gamma \in \mathbb{T}_{env}$, context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, and execution states $(\mu', \iota', \sigma, \eta) :: S, (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$, the transitions $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^{*} (\mu'', \iota'', \sigma, \eta) :: S$ are such that:

$$\mu'.\text{s} = a :: s = \mu''.\text{s},$$

with $\mu.\text{s} = s$

$$\mu'.\text{gas} = \mu.\text{gas} - 12$$

$$\mu''.\text{gas} = \mu.\text{gas} - 6$$

$$\mu'.\text{pc} = \mu.\text{gas} + 2 + 2 \times n$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{gas} + 1 + n$$

□

Proposition 29.

$$\mathsf{OP}' \quad \prec_{opt} \quad \begin{array}{c}\mathsf{SWAP1}\\ \mathsf{OP}\end{array}$$

$\mathsf{OP}, \mathsf{OP}' \in \{\mathsf{LT}, \mathsf{GT}\}$

Proof. The action sequence $\pi = \mathsf{SWAP1}\ \mathsf{OP}$ rearranges the top two elements of the execution stack, given the context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, removing them afterwards and placing on the top of the stack either `0x01` or `0x00`, according to the truth value of $s_1 < s_0$ ($s_1 > s_0$) for $\mathsf{OP} = \mathsf{LT}$ ($\mathsf{OP} = \mathsf{GT}$, respectively) and $\mu.\mathsf{s} = s_0 :: s_1 :: s$.

As a consequence, an action sequence that would reduce the gas expenses by 3 units consists of applying the reverse operation of OP, i.e., if $\mathsf{OP} = \mathsf{LT}$, then $\pi' = \mathsf{OP}' = \mathsf{GT}$, and the reverse for $\mathsf{OP} = \mathsf{GT}$.

The $\mathsf{SWAP1}$ opcode only rearranges elements in the stack, which means the premises for the successful execution of π and π' are identical (the stack must contain at least two elements). Therefore, the successful execution of π implies the same result for π'.

Given transaction environment $\Gamma \in \mathbb{T}_{env}$, the execution of both action sequences π and π' with the context $\mathcal{C}$, $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}{}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}{}^{*} (\mu'', \iota'', \sigma, \eta) :: S$, for the execution states $(\mu', \iota', \sigma, \eta) :: S,\ (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$, alter the execution state in the following way:

$$\mu'.\mathsf{s} = (s_1 > s_0) :: s = \mu''.\mathsf{s}$$

$$(\mu'.\mathsf{s} = (s_1 < s_0) :: s = \mu''.\mathsf{s})$$

if $\mathsf{OP} = \mathsf{LT}$ and $\mathsf{OP}' = \mathsf{GT}$ ($\mathsf{OP} = \mathsf{GT}$ and $\mathsf{OP}' = \mathsf{LT}$, respectively) and for $\mu.\mathsf{s} = s_0 :: s_1 :: s$.

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 3 - 3$$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 3$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 2$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{pc} + 1$$

□

Proposition 30.

$$\begin{array}{c}\mathsf{POP}\\ \mathsf{PUSH1}\ \mathtt{0x00}\end{array} \quad \prec_{opt} \quad \begin{array}{c}\mathsf{PUSH1}\ \mathtt{0x00}\\ \mathsf{OP}\end{array}$$

OP $\in$ {MUL, AND}

Proof. This optimization is associated with the fact that zero is the absorbent element for integer multiplication and that a logical conjunction of a given boolean value with False is always False, i.e., the result placed on top of the execution stack, given the context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, after the execution of $\pi =$ PUSH1 `0x00`
OP, will surely be `0x00`. Accordingly, a less expensive action sequence would be $\pi' =$ POP PUSH1 `0x00`, as it would save one unit of gas.

The premises for the execution of the optimized sequence, excluding the gas requirements, are identical to the non-optimized sequence, and correspond only to the presence of at least one element on the stack. Therefore the successful execution of the later implies the same for the first.

Given the transaction environment $\Gamma \in \mathbb{T}_{env}$, the execution of both action sequences π and π' with the context $\mathcal{C}$, $\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^{*} (\mu'', \iota'', \sigma, \eta) :: S$, for $(\mu', \iota', \sigma, \eta) :: S,\ (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$, alter the execution state in the following way:

$$\mu'.\mathsf{s} = \mathsf{0x00} :: s = \mu''.\mathsf{s},$$

with $\mu.\mathsf{s} = s_0 :: s$.

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 3 - 3$$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 2 - 3$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 2 = \mu''.\mathsf{pc}$$

□

Proposition 31.

$$\begin{array}{ccc} & & \mathsf{ISZERO} \\ \mathsf{ISZERO} & \prec_{opt} & \mathsf{ISZERO} \\ & & \mathsf{ISZERO} \end{array}$$

OP $\in$ {MUL, AND}

Proof. By successively applying three opcodes ISZERO, the output would consist of the equivalent of the double negation of the output of a single opcode ISZERO: "the element on the top of the stack corresponds to `0x00`", thus corresponding to a single application of ISZERO.

As the optimized action sequence corresponds to the first action of the original sequence, the premises for the successful execution are included in the premises for the non-optimized sequence. Consequently, the successful execution of the original sequence implies the successful execution of the optimized sequence as well.

In the presence of the transaction environment $\Gamma \in \mathbb{T}_{env}$, the execution of both action sequences $\pi =$ ISZERO ISZERO ISZERO and $\pi' =$ ISZERO given the context $\mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S$, represented by

$\Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}^{*} (\mu', \iota', \sigma, \eta) :: S$ and $\Gamma \vDash \mathcal{C}[\pi'] \xrightarrow{\pi'}^{*} (\mu'', \iota'', \sigma, \eta) :: S$, for $(\mu', \iota', \sigma, \eta) :: S,\ (\mu'', \iota'', \sigma, \eta) :: S \in \mathbb{S}$, alter the execution state in the following way:

$$\mu'.\mathsf{s} = \mathsf{iszero}(s_0) :: s = \mu''.\mathsf{s},$$

with $\mu.\mathsf{s} = s_0 :: s$.

$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 3 \times 3$$

$$\mu''.\mathsf{gas} = \mu.\mathsf{gas} - 3$$

$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 3$$

$$\mu''.\mathsf{pc} = \mu.\mathsf{pc} + 1,$$

where $\texttt{iszero}(s_0)$ corresponds to the element placed on the top of the stack after execution of ISZERO, i.e., `0x01`, if $s_0 =$ `0x00`, and `0x00`, otherwise. $\square$

Chapter 4

Turing Completeness for a Modified EVM and Control Flow Decision Problems

The notion of computability is intrinsically related to the respective computational model. On his answer to the *Entscheidungsproblem* [41], Alan Turing introduced the notion of computable numbers in order to prove the undecidability of first-order logic. This notion of computable numbers made place for the concept computable functions possible as procedures which can be encoded in what latter came to be known as Turing machines.

In a separate contemporary work by Alonzo Church [13], the author arrived to an analogous conclusion for the functional calculus. Church presented the "effectively computable" functions as the functions which are λ-definable.

Besides these two notions of computable functions, which were proven to be equivalent, the Unlimited Register Machine (URM), proposed by Shepherdson and Sturgis in [38], also constitutes a compatible computational model for defining the set of computable functions.

Due to similarities between the EVM bytecode and the URM instructions, the URM appeared to be the most reasonable choice of computational model to use with the goal of proving the Turing completeness of the EVM.

Nevertheless, some structural limitations of the EVM revealed to be incompatible with the result we intended to prove. Thus, a modified version of the EVM, whose construction shall be clarified over this chapter, was generally specified in order to fulfill the requirements of the proof.

4.1 Unlimited Register Machine

The representation of the Unlimited Register Machine presented by Cutland in [14] and the proof of Turing completeness from [33] will guide the various topics that will be discussed in this section.

Definition 15. *A URM consists of an infinite set of records:*

$$R_1, R_2, R_3, R_4, \dots$$

having as states space -$\Omega = \{\omega \mid \omega$ is 0 almost everywhere$\}$- and each state consists of a sequence $\omega = (\omega_1, \omega_2, \omega_3, \omega_4, \dots)$, with $\omega_n = \omega(n)$ matching the contents of the registry R_n (for $n \in \mathbb{N}$ and $n > 0$).

URM computable functions are defined by sequences of instructions designated by programs.

Definition 16. *A URM program $P = I_1, I_2, \dots, I_s$ is a sequence of instructions belonging to one of the four categories:*

- *Zero instruction $Z(n) : \omega \mapsto \omega'$, where:*

$$\omega'(i) := \begin{cases} 0, & \textit{if } i = n, \\ \omega(i), & \textit{otherwise} \end{cases}$$

- *Successor instruction $S(n) : \omega \mapsto \omega'$, where:*

$$\omega'(i) := \begin{cases} \omega(n) + 1, & \textit{if } i = n, \\ \omega(i), & \textit{otherwise} \end{cases}$$

- *Transfer instruction $T(m, n) : \omega \mapsto \omega'$, where:*

$$\omega'(i) := \begin{cases} \omega(m), & \textit{if } i = n, \\ \omega(i), & \textit{otherwise} \end{cases}$$

- *Jump instruction $J(m, n, q) : \omega \mapsto \omega'$, which does not change the records, only the order that the instructions are executed, i.e., $\omega' = \omega$. If $\omega(m) = \omega(n)$, the next instruction to execute corresponds to the instruction I_q of program P (for $q < |P|$), otherwise the execution proceeds to the next statement (if it is being executed I_k, I_{k+1} will be the next instruction, taking $|P| \geq k + 2$, or it ends).*

The input of P is a sequence of natural numbers $a_1, a_2, a_3, \dots$, which are placed in the respective registers $R_1, R_2, R_3, \dots$, thus producing a URM state referred to as initial configuration.
The output of P is stipulated to be stored in the R_1 register of the final configuration.
The sets of all URM instructions and programs are denoted by $\mathscr{I}$ and $\mathscr{P}$, respectively.

The computation of a URM program P is performed starting at I_1 and proceeding sequentially, i.e., unless for the case of the current instruction, I_k, pertaining to the jump type, $J(m, n, q)$, the next instruction will be I_{k+1} if $k + 1 \leq |P| = s$. Halting is described when the address of the instruction to execute next is greater than s for $P = I_1, I_2, \dots, I_s$.

Definition 17. *Let $a_1, a_2, a_3, \dots$ be a sequence of natural numbers, and P a URM program.*

- *The computation of the given sequence by P is denoted by $P(a_1, a_2, a_3, \ldots)$.*
- *For the inputs that contain only a finite number of entries different than zero $a_1, a_2, \ldots, a_n$, $P(a_1, a_2, \ldots, a_n)$ is used to refer to $P(a_1, a_2, \ldots, a_n, 0, 0, \ldots)$.*
- *A computation that halts is denoted by $P(a_1, a_2, a_3, \ldots) \downarrow$. If b is in R_1 in the final configuration, then $P(a_1, a_2, a_3, \ldots) \downarrow b$.*
- *A computation which never stops is denoted by $P(a_1, a_2, a_3, \ldots) \uparrow$.*

In order to define the concatenation of programs, some changes must be operated on the jump instructions, so that the integrity of the purpose of the program is preserved.

Definition 18. *A URM program $P = I_1, I_2, \ldots, I_s$ is in standard form if, for every jump instruction $J(m, n, q)$, $q \leq s + 1$ holds.*

Lemma 1. *For very URM program P, there exists a program in the standard form P^* such that, for every $n > 0$ and $a_1, a_2, \ldots, a_n, b \in \mathbb{N}$, the following equivalence holds:*

$$P(a_1, a_2, \ldots, a_n) \downarrow b \Leftrightarrow P^*(a_1, a_2, \ldots, a_n) \downarrow b$$

Definition 19. *Given two URM programs, $P_1 = I_1, I_2, \ldots, I_{s_1}$ and $P_2 = I'_1, I'_2, \ldots, I'_{s_2}$ in the standard form, the concatenation, denoted by P_1P_2, consists of the program $I_1, I_2, \ldots, I_{s_1+s_2}$, being obtained by replacing the jump instructions of P_2, $J(m, n, q)$, by $J(m, n, s_1 + q)$*

4.2 Modified Ethereum Virtual Machine

Although the most common high-level language that is used for writing smart contracts, Solidity, is Turing complete, the EVM bytecode does not share the same property.

Furthermore, Turing completeness may not be needed for a significant number of contracts, as observed by [27]. This study inspects verified smart contracts accessible through the `Etherscan.io` repository, arriving to the conclusion, after an analysis oriented by the occurrence of regular expressions on smart contracts, that only 6.9% of the contracts exhibit while-loops.

The lack of control over programs derived from the Turing completeness of Solidity lead to the development of the Vyper language, which is deliberately not Turing complete.

On the other hand, reading and writing smart contracts may be facilitated, due to resemblances between Vyper and Python.

To affirm that the EVM is Turing complete implies that it can simulate any function computable by a Turing machine. Therefore, the capacity to perform unbounded arithmetic operations and unlimited storage capacity are some characteristics that should be embedded in the EVM and, as discussed previously, does not exist under the current definition of its constraints.

Proceeding in this fashion, we may observe, as stated before, that the call stack depth is limited to 1024, with every nested call gas limit restricted to $63/64$ of the "parent" call limit.

Furthermore, computations in the EVM are bounded by the element gas. For instance, besides the fact that all transactions are beforehand bounded by a limit of gas to be spent, the overall of every block's operations can not exceed a given gas consumption. This limit can not be greater than the previous block's gas limit increased by a given fraction of it.

Hence, the EVM version that will be proven Turing complete differs from the original in the following aspects:

- The number of entries in the storage of a contract is infinite and all entries are initialized to zero.
- Gas ceases to exist, along with the system information EVM opcodes and conditions for contract execution associated with it.
- The size of the words stored in permanent (smart contracts storage) or volatile storage (memory and execution stack) may be arbitrary.
- Instructions which take items from the execution stack or memory no longer operate with 256-bit items but arbitrarily large bitstrings. The fact that the EVM opcodes which operate on stack elements consist mainly on arithmetic and logical operations, looking up functions which receive indexes and the Keccack-256 hash (whose algorithm operates on words with arbitrary size), makes it possible to induce this alteration.
- All of the PUSHn $(n = 1, \dots, 32)$ instructions are condensed in the instruction PUSH which places an arbitrary length word on the top of the stack. The code for PUSH shall be the same as PUSH1 (`0x60`).
- The encoding of a word $w = b_1 b_2 \dots b_n$ with n bytes is performed by the variable length encoding scheme h given below:
$$h(w) = b'_1 b_1 b'_2 b_2 \dots b'_n b_n,$$
where $b'_i =$ `0x01`, if $i \neq n$, and $b'_i =$ `0x00`, otherwise.
- The previous statement implies that the execution of a PUSH will increase the program counter by $2k + 1$ units, where k is the number of bytes of the argument of the push instruction.

With these broadly defined new restrictions, it will be possible to prove the completeness of the EVM and, thus, we will also be able to prove interesting results regarding the control-flow of smart contracts.

4.3 Translation of URM Instructions

Let us describe the translation of URM instructions into operations of the construction alluded in the previous section for the EVM.

```
00              PUSH 0x00
03              PUSH hex(m)
hex(04 + 2k)    SSTORE
```

Figure 4.1: $Z(m)$ instruction translated to the modified EVM bytecode (`hex(m)` symbolizes the hexadecimal representation of m and `k` corresponds to the respective number of bytes of `hex(m)`)

```
00                        PUSH hex(m)
01                        SLOAD
hex(02 + 2k)              PUSH 0x01
hex(05 + 2k)              ADD
hex(06 + 2k)              PUSH hex(m)
hex(07 + 4k)              SSTORE
```

Figure 4.2: $S(m)$ instruction translated to the modified EVM bytecode (`hex(m)` symbolizes the hexadecimal representation of m and `k` corresponds to the respective number of bytes of its hexadecimal representation)

```
00                    PUSH hex(m)
hex(01 + 2k)          SLOAD
hex(02 + 2k)          PUSH hex(n)
hex(03 + 2k + 2k')    SSTORE
```

Figure 4.3: $T(m, n)$ instruction translated to the modified EVM bytecode (`hex(m)` and `hex(n)` symbolize the hexadecimal representations of m and n, and `k` and `k'` correspond to the respective number of bytes of their hexadecimal representation)

```
00                            PUSH hex(m)
hex(01 + 2k)                  SLOAD
hex(02 + 2k)                  PUSH hex(n)
hex(03 + 2k + 2k')            SLOAD
hex(04 + 2k + 2k')            EQ
hex(05 + 2k + 2k')            PUSH q
hex(06 + 2k + 2k' + 2k'')     JUMPI
          ⋮
          q                   JUMPDEST
```

Figure 4.4: $J(m, n, q)$ instruction translated to the modified EVM bytecode (`hex(m), hex(n)` symbolize the hexadecimal representations of m and n and `k, k', k''` correspond to the respective number of bytes of the hexadecimal representation of m, n and q)

Proposition 32. *Every URM program can be simulated by some modified EVM program.*

Proof. All the URM instructions may be translated to the bytecode, as seen above, thus, the functions that can be written with the URM are also computable by the modified EVM. Preserving functionality is achieved by a re-enumeration of the addresses of the program instructions as the number of instructions needed for the modified EVM to simulate each URM instruction is always greater than one (7 instructions for $J(m, n, q)$, 4 for $T(m, n)$, 5 for $S(m)$ and 3 for $Z(m)$). $\square$

4.4 URM-Computability and Undecidability of Jumps

Before proving results regarding the control flow of modified EVM programs, some notions of computability theory must be introduced, while others, the concept of URM-computable functions, for instance, shall be introduced according to [14].

Definition 20. *A function $f : \mathbb{N}^k \mapsto \mathbb{N}$ is URM-computable if there is a program P that halts for the a all $(a_1, a_2, \ldots, a_k) \in \mathbb{N}^k$ if and only if $f(a_1, a_2, \ldots, a_k)$ is defined.*

The class of URM-computable functions is denoted by $\mathscr{C}$ (for the case of n-ary function, the respective class shall be denoted by $\mathscr{C}_n$).

As the computation of each instruction proceeds in a deterministic manner, when considering initial configurations associated to inputs of the form $a_1, a_2, \ldots, a_n, 0, 0, \ldots$, only one function may be associated to a given URM program P, the function that shall be denoted by $f_P^{(n)}$.

4.4.1 Properties of URM-Computable functions

Definition 21. *A set X is denumerable if there is a bijection $f : X \mapsto \mathbb{N}$. Moreover, X is classified as effectively denumerable if both f and f^{-1} are computable functions.*

Proposition 33. *The sets $\mathscr{I}$, $\mathscr{P}$ are effectively denumerable and the set $\mathscr{C}$ is denumerable.*

Proof. It is reasonably evident that $\mathscr{I}$ is effectively denumerable as the number of types of instructions is finite (four) and they receive a natural number as input.
The explicit bijection between $\mathscr{I}$ and $\mathbb{N}$ is given by:

$$\beta : \mathscr{I} \mapsto \mathbb{N}$$

, such that, for $n \in \mathbb{N},\ n \geq 1$,

$$\beta(Z(n)) = 4(n-1)$$

$$\beta(S(n)) = 4(n-1) + 1$$

$$\beta(T(m,n)) = 4\pi(m-1, n-1) + 2$$

$$\beta(J(m,n,q)) = 4\zeta(m,n,q) + 3,$$

with π and ζ defined as:

$$\pi : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$$

$$\pi(m,n) = 2^m(2^n+1)-1$$

$$\zeta : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$$

$$\zeta(m,n,q) = \pi(\pi(m-1,n-1),q-1)$$

Cutland also provides the inverse of these functions and proves the respective computability.

The analogous result for $\mathscr{P}$ is proven by establishing the bijection $\gamma : \mathscr{P} \mapsto \mathbb{N}$, resorting to the bijections τ and β. Given a program $P = I_1, I_2, \ldots, I_s$,

$$\gamma(P) = \tau(\beta(I_1), \beta(I_2), \ldots, \beta(I_s))$$

with $\tau : \cup_{k>0}\mathbb{N}^k \mapsto \mathbb{N}$ defined as:

$$\tau(a_1, \ldots, a_k) = 2^{a_1} + 2^{a_1+a_2} + 2^{a_1+a_2+a_3} + \ldots + 2^{a_1+a_2+a_3+\ldots+a_k-1} - 1$$

, for $a_1, \ldots, a_k \in \mathbb{N}$
γ^{-1} is also proven to be computable by Cutland and P_n is used to refer $\gamma^{-1}(n)$. $\square$

Definition 22. *For each $a \in \mathbb{N}$ and $n \geq 1$:*

- *$\phi_a^{(n)}$ corresponds to the n-ary function computed by the program P_a.*
- *$W_a^{(n)}$ denotes the domain of $\phi_a^{(n)}$, i.e., $\{(x_1, x_2, \ldots, x_n) |\ P_a(x_1, x_2, \ldots, x_n) \downarrow\}$.*
- *$E_a^{(n)}$ denotes the range of $\phi_a^{(n)}$.*

Definition 23. *For each $n \in \mathbb{N}$ and $n \geq 1$, the universal function for n-ary functions is the $(n+1)$-ary function $\psi_U^{(n)}$ defined by:*

$$\psi_U^{(n)}(e, x_1, \ldots, x_n) = \phi_e^{(n)}(x_1, \ldots, x_n)$$

Lemma 2. *For each $n \in \mathbb{N}$ and $n \geq 1$, the universal function $\psi_U^{(n)}$ is computable.*

4.4.2 Decision Problems

A decision problem over natural numbers consists of the question of whether they pertain to a set of natural numbers which enjoys a certain property, i.e., for a given $n \in \mathbb{N}$, $C \subseteq \mathbb{N}^n$ and with $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{N}^n$ to answer to whether $\mathbf{x} \in C$ or not.

Definition 24. *The characteristic function of the set C of a decision problem $\mathcal{P}$ is given by the function $\chi_C : \mathbb{N}^n \mapsto \{0,1\}$, such that:*

$$\chi_C(\mathbf{x}) := \begin{cases} 1, & \textit{if } \mathbf{x} \in C \\ 0, & \textit{c.c.} \end{cases}$$

A decision problem is said to be decidable if the characteristic function of its respective set is computable.

The problem reduction technique consists of, given two decision problems $\mathcal{P}_1$ and $\mathcal{P}_2$ with respective delimiters of the problem space n_1 and n_2 and sets C_1 and C_2, finding a computable map $s : \mathbb{N}^{n_1} \mapsto \mathbb{N}^{n_2}$ that entails the following equivalence:

$$w \in C_1 \;\Leftrightarrow\; s(w) \in C_2$$

The following results consist of a particular instance of the halting problem that is proven undecidable by the diagonal method of construction of functions, a procedure inspired by Cantor's work, a formulation for the halting problem and the printing problem for a URM program P. All the referred topics are approached by Nigel Cutland in [14], a perspective that this work will follow.

Proposition 34. *The problem of, given $x \in \mathbb{N}$, whether $x \in W_x$ is undecidable.*

Proof. As it was already mentioned, the proof of this result proceeds through the construction of functions resorting to the diagonal method with the goal of establishing a contradiction.

Let us suppose the characteristic function related to this decision problem is computable.

$$\chi_{W_x}(\mathbf{x}) := \begin{cases} 1, \text{ if } \mathbf{x} \in W_x \\ 0, \text{ c.c.} \end{cases}$$

Let $g : \mathbb{N} \mapsto \{0\}$ be the function defined by:

$$g(\mathbf{x}) := \begin{cases} 0, \text{ if } \mathbf{x} \notin W_x \\ \text{undefined , if } \mathbf{x} \in W_x \end{cases}$$

The computability of χ_{W_x} entails the computability of g, thus, implying that there must be some $m \in \mathbb{N}$ such that $g = \phi_m$. Therefore, the contradiction is established as it is shown by the equivalence: $m \in W_m \Leftrightarrow P_m(m) \downarrow 0 \Leftrightarrow \phi_m(m) = 0 \Leftrightarrow m \notin W_m$. □

Proposition 35. *Given $x, y \in \mathbb{N}$, the problem of determining whether $\phi_x(y)$ is defined (or equivalently $P_x(y) \downarrow$) is undecidable.*

Proof. It is trivial to see that the decidability of this problem entails the decidability of the previous one (known to be undecidable), thus, it is also classified as undecidable. □

Proposition 36. *Given $x, y \in \mathbb{N}$, the problem of determining whether $y \in E_x$ (or equivalently if $y \in range(\phi_x)$) is undecidable.*

Theorem 1. *Given a modified EVM program $M = e_1, e_2, \ldots, e_n$, such that there exists at least one occurrence of $e_i =$ JUMPI ($1 \leq i \leq n$), the problem of determining whether a JUMPI instruction successfully proceeds with the jump to the code location a is undecidable.*

Proof. Bearing in mind the previous result, we shall reduce the present problem to it, thus proving the intended result.

With the intention of performing a proof by *reductio ad absurdum*, let us suppose the problem of knowing if a JUMPI instruction is decidable by a decider algorithm $\mathcal{J}$.

Given a URM program $P = I_1, I_2, \ldots, I_u$ in the standard form, it is possible to translate P to the modified EVM bytecode (the resulting program shall be named M_P with s bytes).

By appending to the code of M_P the instructions:

```
hex(s + 1)    JUMPDEST
hex(s + 2)    PUSH  hex(s + 10 + 2k + 2k')
hex(s + 3 + 2k)  PUSH  b
hex(s + 4 + 2k + 2k')    PUSH  0x01
hex(s + 6 + 2k + 2k')    SLOAD
hex(s + 7 + 2k + 2k')    EQ
hex(s + 8 + 2k + 2k')    JUMPI
hex(S + 9 + 2k + 2k')    STOP
hex(s + 10 + 2k + 2k')   JUMPDEST,
```

where $\mathtt{k}'$ corresponds to the number of bytes of b and $\mathtt{k}$ is the smallest positive integer corresponding to the number of bytes of the hexadecimal representation of $\mathtt{hex}(\mathtt{s} + 10 + 2\mathtt{k} + 2\mathtt{k}')$ (the hexadecimal representation of $s + 10 + 2k + 2k'$).

As a consequence, by applying algorithm J, one would be able to affirm for the program M_P if the JUMPI at the code location $\mathtt{s} + 8 + 2\mathtt{k} + 2\mathtt{k}'$ eventually executes the jump to the address $\mathtt{s} + 10 + 2\mathtt{k} + 2\mathtt{k}'$, thus, also revealing whether b belongs in E_P. $\square$

Theorem 2. *Given a modified EVM program $M = e_1, e_2, \ldots, e_n$, such that there exists at least one occurrence of $e_i =$ JUMP $(1 \leq i \leq n)$, the problem of determining whether a JUMP instruction successfully proceeds with the jump to the code location a is undecidable.*

Proof. The reasoning for the proof of intended result consists of a reduction of the halting problem, which is undecidable by proposition 35, to the JUMP decision problem.

With the intention of performing a proof by *reductio ad absurdum*, let us suppose the problem of knowing if a JUMP instruction jumps to a given code location is decidable by a decider algorithm $\mathcal{J}'$.

Given a URM program $P' = I'_1, I'_2, \ldots, I'_u$ in the standard form, it is possible to translate P' to the modified EVM bytecode (the resulting program shall be named $M_{P'}$ with s bytes).

By appending to the code of $M_{P'}$ the instructions:

```
hex(s + 1)   JUMPDEST
hex(s + 2)   PUSH hex(s + 4 + 2k)
hex(s + 3 + 2k) JUMP
hex(s + 4 + 2k) JUMPDEST
hex(s + 5 + 2k) PUSH 0x00
hex(s + 6 + 2k) PUSH 0x00
hex(s + 7 + 2k) RETURN,
```

where $\mathtt{k}$ the smallest positive integer corresponding to the number of bytes of $\mathtt{hex}(\mathtt{s}+4+2\mathtt{k})$ (the hexadecimal representation of $s+4+2k$).

As a consequence, by applying algorithm J' to find out if the JUMP at the address $\mathtt{s}+3+2\mathtt{k}$ executes successfully, one would be able to affirm if the program P' terminates. If the JUMP at the address $\mathtt{s}+3+2\mathtt{k}$ eventually executes the jump to the code location $\mathtt{s}+4+2\mathtt{k}$, then the program counter must have assumed the values $\mathtt{s}+1$ and $\mathtt{s}+2$, thus ensuring termination. $\square$

Chapter 5

Smart Contract Optimizer

Smart contracts as introduced by Nick Szabo [40] consist mainly on a strong alternative to the burden of placing trust on third-parties on the agreement of contract terms. Contract clauses are embedded in hardware and software, thus eliminating cost, which might have been prohibitive.

In other words, Szabo proposes that the main purpose of each contract, to honer promises agreed to in the "meeting of the minds", shall be enforced by computerized protocols.

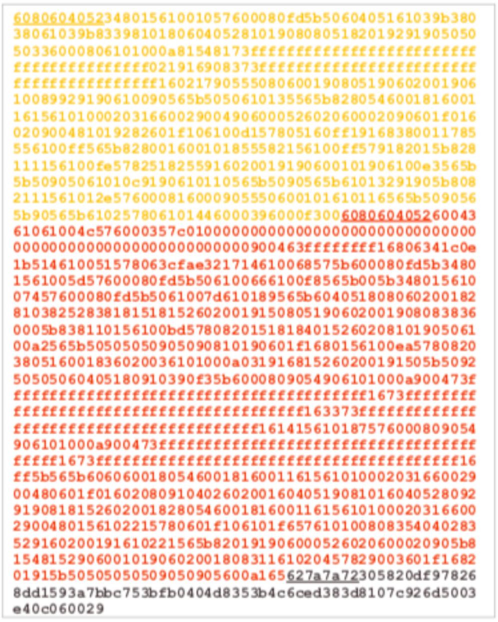
In this model, observability is a key feature that contracts must exhibit. Principals must be able to prove to third parties and other principals the accomplishment of the agreed terms. Furthermore, the functionality and implications of these protocols is to be disclosed between parties beforehand.

Ethereum smart contracts embody this idea as all contracts obey the respective code, which is immutable once the miners run the respective contract creation transaction with the given deployment code, and remain available to the public in the blockchain.

Furthermore, the execution of these contracts is deterministic and can be proven by any individual, thus fulfilling the requirements stated by Szabo.

5.1 Smart Contract Structure

A compiled contract consists of three components: the loader code, the runtime code and the swarm hash. The first corresponds to the code that miners run when processing the contract creation transaction. As a result the second is appended to the blockchain and is the code that actually runs when calling the contract. Concluding, the swarm hash is the hash of the root of the Merkle-Patricia Tree storing the contract account contents.



```
608060405234801561001057600080fd5b5060405161039b380
38061039b83398101806040528101908080518201929190505050
50336000806101000a81548173fffffffffffffffffffffffff
fffffffffffffff021916908373ffffffffffffffffffffffff
ffffffffffffffff16021790555080600190805190602001906
1008992919061009056 5b5050610135565b828054600181600 1
16156101000203166002900490600052602060002090601f016
02090048101928260 1f106100d157805160ff1916838001178 5
556100ff565b82800160010185558215610 0ff579182015b828
111156100fe578251825591602001919060010190610 0e3565b
5b5090506101 0c9190610110565b5090565b61013291905b808
2111561012e576000816000905550600101610116565b509056
5b90565b6102578061014460003960 00f3006080604052600 43
61061004c576000357c0100000000000000000000000000000000
0000000000000000000000000900463ffffffff16806341c0e
1b514610051578063cfae3217146100685 75b600080fd5b3480
1561005d57600080fd5b506100666100f8565b005b348015610
07457600080fd5b5061007d610189565b6040518080602001 82
810382528381815181526020019150805190602001908083836
0005b838110156100bd578082015181840152602081019050 61
00a2565b505050509050908101906 01f1680156100ea578082 0
3805160018360200361010 00a031916815260200191505b5092
5050506040518091039 0f35b6000809054906101000a900473f
fffffffffffffffffffffffffffffffffffffff1673ffffffff
ffffffffffffffffffffffffffffffff163373fffffffffffff
fffffffffffffffffffffffffff161415610187576000809054
906101000a900473fffffffffffffffffffffffffffffffffff
fffff1673ffffffffffffffffffffffffffffffffffffffff16
ff5b565b6060600180546001816001161561010002031660029
00480601f016020809104026020016040519081016040528092
919081815260200182805460018160011615610100020316600
290048015610221578060 1f106101f6576101008083540402 83
529160200191610221565b8201919060005260206000209 05b8
15481529060010190602001808311610204578290036 01f1682
01915b505050505050905090 5600a165627a7a72305820df97826
8dd1593a7bbc753bfb0404d8353b4c6ced383d8107c926d5003
e40c060029
```

Figure 5.1: Smart Contract Bytecode: Loader Code (yellow), Runtime Code (red) and Swarm Hash (black) from [42]

Smart contracts may be defined as sequential programs, i.e., sets of instruction of the form $P = i_0 i_1 \dots i_n$, with each $i_j, j = 0, \dots, n$ denoting an instruction corresponding to EVM opcodes, which can be organized in a block sequence.

The execution of a contract runtime code always starts at the code location `0x00` and the first portion of the code can be abstracted in a dispatcher function, comparing the first four bytes with the function signatures associated with the contract.

A function signature consists of the first four bytes of the Keccak hash of the function name.

$$\texttt{Keccak}(kill()) = 41c0e1b5eba5f1ef69db2e30c1ec7d6e0a5f3d39332543a8a99d1165e460a49e$$

For the example above the signature of the function `kill`$()$ would be $41c0e1b5$.

Interacting with smart contracts is achieved by specifying the function signature of the intended procedure to execute followed by the respective arguments in the call data.

Aiming to scrutinize the control flow of a contract, the characterization of its structure shall be provided, in consonance with the detailed structure proposed by Beatriz in [45].

The instructions that constitute the contracts bytecode may be grouped in blocks, which are delimited by either the opcodes that operate on the program counter (JUMP/JUMPI and JUMPDEST) or by termination opcodes (STOP, RETURN, REVERT and INVALID).

Definition 25. *A bytecode block b_i of a program $P = i_0, i_1, \dots, i_n$ is denoted by a sequence of instructions $b_j = b_{j,0} \dots b_{j,k}$ such that:*

- $b_{j,0} = i_0$ *and* $j = 0$*, or* $b_{j,0} = \text{JUMPDEST}$*, or* $b_{j,0} = i_{l+1}$*, with* $i_l = \text{JUMPI}$*.*
- $b_{j,k} = i_n$ *or* $b_{j,k} \in \{\text{JUMP}, \text{JUMPI}, \text{STOP}, \text{RETURN}, \text{REVERT}\}$*, or* $b_{j,k} = \text{INVALID}$*, or* $b_{j,k} = i_l$*, with*

$i_{l+1} =$ JUMPDEST.

A directed graph is the most common representation for the collective of blocks that constitute a contract, as a whole. In this fashion, blocks define the vertices and the edges are identified resorting to a set of established rules.

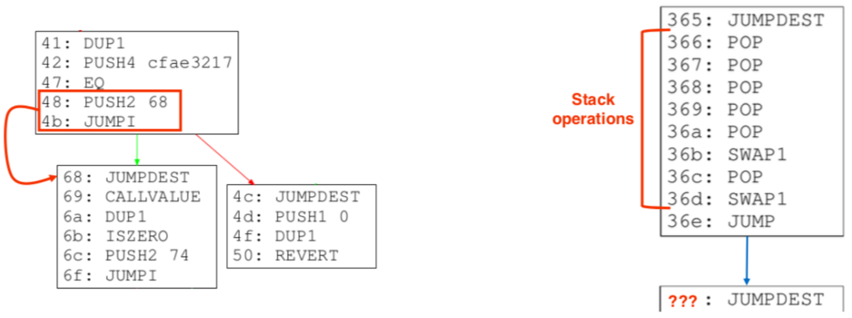
Definition 26. *The control flow graph of a contract is a directed graph $G = \langle V, E \rangle$, with V corresponding to the set of vertices and $E \subseteq V^2$ to the set of edges. An edge $(a, b) \in E$ is such that:*

- $a =$ JUMP *and* $b =$ JUMPDEST*, with the offset of* b *in the range of jump targets of* a*.*
- $a =$ JUMPI *and* $b =$ JUMPDEST*, with the offset of* b *in the range of jump targets of* a*.*
- $a =$ JUMPI *and the offset of* b *is next to the one of* a*.*
- $b =$ JUMPDEST *and* a *is the instruction preceding* b*, with* $a \notin \{$JUMP, STOP, RETURN, REVERT, INVALID$\}$*.*

5.2 Control Flow Graphs for Smart Contracts

Although the result on the previous set is proven for the case of the modified EVM and that Turing Completeness can only be observed in this construction as it differs from the original version in the unlimited storage feature, is hard to retrieve the targets for each jump instruction in a real contract.

The difficulty resides in the fact that some JUMP or JUMPI are not preceded by a PUSH instruction. These jumps are designated by dynamic jumps, while for the converse case, the commonly used term is static jumps.



```
41: DUP1
42: PUSH4 cfae3217
47: EQ
48: PUSH2 68
4b: JUMPI
```

```
68: JUMPDEST
69: CALLVALUE
6a: DUP1
6b: ISZERO
6c: PUSH2 74
6f: JUMPI
```

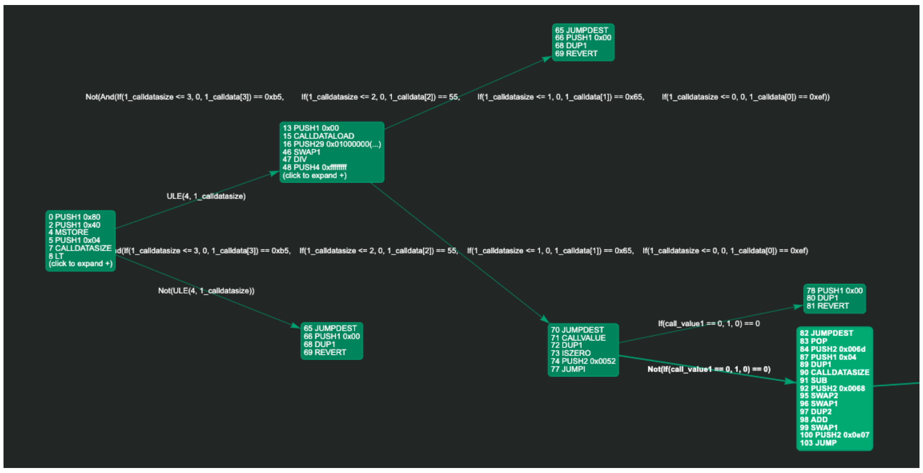
```
4c: JUMPDEST
4d: PUSH1 0
4f: DUP1
50: REVERT
```

```
365: JUMPDEST
366: POP
367: POP
368: POP
369: POP
36a: POP
36b: SWAP1
36c: POP
36d: SWAP1
36e: JUMP
```

```
??? : JUMPDEST
```

Figure 5.2: Static and Dynamic jumps from [42]

In order to obtain the control flow graph associated with the bytecode of a smart contract, we resorted to the use of the Mythril tool [32].


```
0 PUSH1 0x80
2 PUSH1 0x40
4 MSTORE
5 PUSH1 0x04
7 CALLDATASIZE
8 LT
(click to expand +)
```

```
13 PUSH1 0x00
15 CALLDATALOAD
16 PUSH29 0x01000000(...)
46 SWAP1
47 DIV
48 PUSH4 0xffffffff
(click to expand +)
```

```
65 JUMPDEST
66 PUSH1 0x00
68 DUP1
69 REVERT
```

```
65 JUMPDEST
66 PUSH1 0x00
68 DUP1
69 REVERT
```

```
70 JUMPDEST
71 CALLVALUE
72 DUP1
73 ISZERO
74 PUSH2 0x0052
77 JUMPI
```

```
78 PUSH1 0x00
80 DUP1
81 REVERT
```

```
82 JUMPDEST
83 POP
84 PUSH2 0x006d
87 PUSH1 0x04
89 DUP1
90 CALLDATASIZE
91 SUB
92 PUSH2 0x0068
95 SWAP2
96 SWAP1
97 DUP2
98 ADD
99 SWAP1
100 PUSH2 0x0e07
103 JUMP
```
Not(And(If(1_calldatasize <= 3, 0, 1_calldata[3]) == 0xb5, If(1_calldatasize <= 2, 0, 1_calldata[2]) == 55, If(1_calldatasize <= 1, 0, 1_calldata[1]) == 0x65, If(1_calldatasize <= 0, 0, 1_calldata[0]) == 0xef))
ULE(4, 1_calldatasize)
nd(If(1_calldatasize <= 3, 0, 1_calldata[3]) == 0xb5, If(1_calldatasize <= 2, 0, 1_calldata[2]) == 55, If(1_calldatasize <= 1, 0, 1_calldata[1]) == 0x65, If(1_calldatasize <= 0, 0, 1_calldata[0]) == 0xef)
Not(ULE(4, 1_calldatasize))
If(call_value1 == 0, 1, 0) == 0
Not(If(call_value1 == 0, 1, 0) == 0)

Figure 5.3: Portion of a control flow graph of a smart contract obtained with the Mythril command line tool

Mythril, similarly to Oyente, relies on symbolic execution of smart contracts allied to the use of an SMT solver, Z3, in order to evaluate the viable paths throughout an execution. The module responsible for the symbolic execution is called LASER-Ethereum and the source code is available at [31].

LASER takes into account in each step information relative to the current execution state and the "path formula", a conjunction of predicates involving variables relative to elements pertaining to the state or stack and memory elements which are appended through each state transition.

Bearing in mind the specification of the execution model provided in the Ethereum Yellow Paper [44], the notion of state held by the LASER module consists of the aggregate of three components: the global state, σ, the machine state, μ, and the execution environment, I.

The global state, as stated before, consists of a mapping between addresses and account contents, whose changes are performed via the execution of transactions. In spite of the hash of the root of a Merkle-Patricia tree, the LASER implementation of the `codeHash` parameter of an account contains the code itself under the format of a disassembly object obtained by Mythril.

Regarding the machine state, as it has been already seen for the case of the semantics specified by Grishchenko *et al.*, it is expressed by a tuple $\mu = (g, pc, m, i, s)$, with g standing for the gas available for the execution, pc for the program counter, m for the execution memory array, i for the value corresponding to the number of active words in memory and s to the execution stack.

For the definition of the execution environment, the elements contained in this tuple must be introduced, as it is done below.

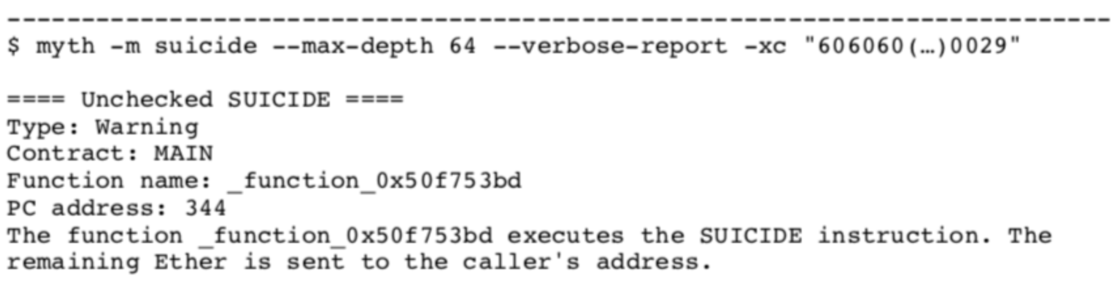
- I_a, the address of the account that owns the executing code.
- I_o, the sender address of the transaction that originated the execution.
- I_p, the price of gas in the transaction that originated the execution.
- I_d, the input byte array for the execution. If the execution agent is a transaction, I_d is the transaction data.
- I_s, the address of the account that caused the code execution. If the execution agent is a transaction, I_s is the transaction sender.

- I_v, the value in Wei passed to this account as part of the procedure that execution belongs to. If the execution agent is a transaction, I_v is the transaction value.
- I_b, the byte array represented by the machine code to be executed. - I_H, the block header of the present block.
- I_e, the depth of the present message call or contract-creation (i.e., the number of CALLs or CREATEs being executed).

As a result of its capabilities for searching the space of possible execution states, Mythril has proven to be very effective in determining violations of safety properties such as possible unprotected access to deploying a transaction in which the contract successfully executes the SUICIDE command.

Terrible and irreversible effects may be the outcome of exploiting these contract vulnerabilities. As an example, the Parity's official multi-signature wallet library contract was eliminated through the successive execution of two transactions. First, the user gained ownership of the contract by initializing a constructor and then send another transaction invoking the `kill()` function.

This contract was implemented in an ill manner, as the developers ignored the existence of proven safe Solidity code libraries, opting for the adaptation of an already deployed similar contract, thus exposing an error with losses estimated in 150 million US dollars [12].



```
--------------------------------------------------------------------------
$ myth -m suicide --max-depth 64 --verbose-report -xc "606060(…)0029"

==== Unchecked SUICIDE ====
Type: Warning
Contract: MAIN
Function name: _function_0x50f753bd
PC address: 344
The function _function_0x50f753bd executes the SUICIDE instruction. The
remaining Ether is sent to the caller's address.
```

Figure 5.4: SUICIDE debug performed by the command line Mythril tool [31]

Similarly to the case of detecting eventual SUICIDE instructions in the code, Mythril is equipped with a detection module for scrutinizing possible Ether transfers, `ether_send.py`. It works by signaling CALL instructions, the procedure steps described in [32] are stated below. For every state in which $I_b[\mu_{pc}] =$ CALL:

1. Determine whether the call value is greater than zero.
2. Check the target stack address.
3. For every storage constraint on the node containing the CALL, search for SSTORE instructions that may allow writing to the storage slot.
4. Attempt to satisfy the state's path formula.
5. Report a potential issue if the address is user-supplied, the path formula can be satisfied, and either there are no storage constraints or there are potential paths to writing the respective storage locations.

5.3 Implementation and Results

To operate on Ethereum smart contracts poses several challenges. Although Ethereum is a public blockchain, accessing all the existing contracts is a difficult task, only doable by the network nodes or through parsing through all the transactions.

The constant change of the system makes it hard to track the state of the system at any given moment.

Acquiring the addresses was achieved through parsing of the Etherscan repository. The dataset that was downloaded from Etherscan consists of a small set of over two thousand verified contract addresses.

The Mythril version used for building the control flow graph of contracts and fetching the corresponding bytecode was `Mythril v0.21.3`.

The implemented tool is composed of two components:

- Contract fetcher and graph constructor - based on the use of the Mythril tool, it downloads the code and generates the control flow graph of the contract.

- Optimizer - responsible for parsing the code of each contract in order to find the gas costly patterns, replacing them with the corresponding optimized version.

For testing, we used a simplified version of the tool, which assumes the dynamic jumps only jump to code locations whose hexadecimal representation was placed on the stack by a PUSH instruction, and that all the PUSH instructions which push the hexadecimal representation of JUMPDEST code locations only contribute to dynamic jumps performed by JUMP/JUMPI.

The obtained results are as shown below:

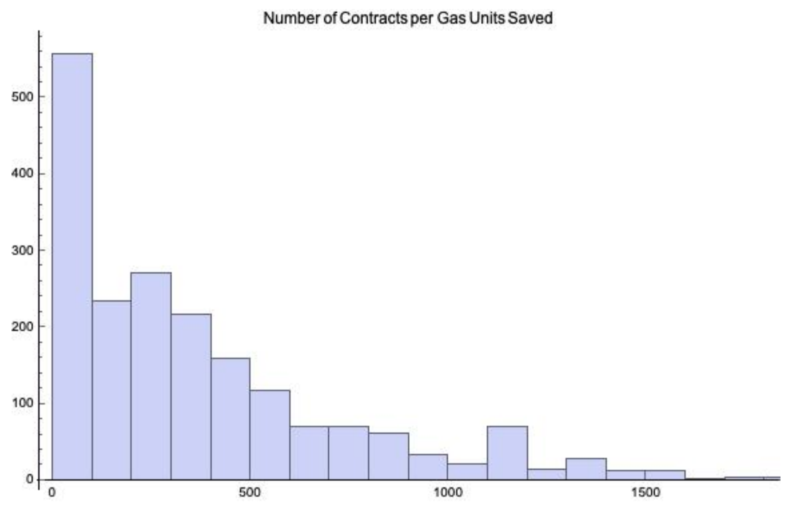


Figure 5.5: Number of contracts per gas saved from the optimization experiment

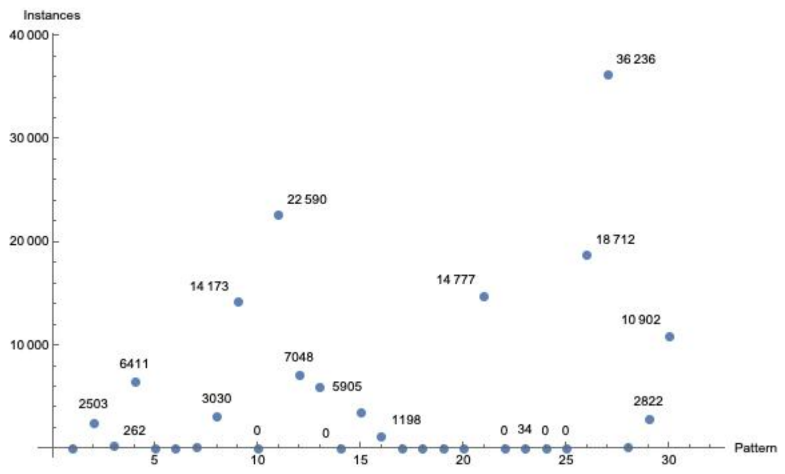


Figure 5.6: Number of instances of each bytecode pattern found throughout the optimization experiment

Chapter 6

Conclusions

6.1 Achievements

We introduce and analyze a series of gas-costly EVM bytecode patterns with the respective optimization, and formally discuss these optimizations and others found in the literature within the context of the EVM semantics.

In order to maintain program integrity, we needed to ensure that the instructions that were responsible for determining the targets of the jump instructions in the code are corrected, after parsing the code and replacing instruction sequences with instances of our list of optimizations. However, for most cases, this task proves to be impossible to attain only with static analysis of contracts.

By studying its features, we found that the EVM itself is not Turing Complete. Although, the state space of a program may be unfeasible to work on as the EVM is referred as *quasi*-Turing Complete by the Ethereum co-founder Gavin Wood.

We prove that, if we remove the gas restrictions, the bounding on the size of words the EVM operates on and with unlimited storage space, the Turing Completeness can be observed and it becomes undecidable to determine the target of jump instructions.

6.2 Future Work

Symbolic execution opens doors for advances on the subject of security of Ethereum smart contracts.

Additionally, the optimization of Ethereum smart contracts may be studied more deeply by either discovering new gas-costly through characterization of the contracts behavior at each code block, or reordering blocks and deletion of irrelevant operations, among other techniques.

Bibliography

[1] Ethash · ethereum/wiki wiki. `https://github.com/ethereum/wiki/wiki/Ethash`. (Accessed on 09/04/2019).

[2] Keccak team. `https://keccak.team/keccak.html`. (Accessed on 09/03/2019).

[3] Solidity documentation. `https://buildmedia.readthedocs.org/media/pdf/solidity-zh/stable/solidity-zh.pdf`. (Accessed on 11/12/2019).

[4] Ethereum (ether) historical prices. `https://etherscan.io/chart/etherprice`. (Accessed on 07/09/2019).

[5] Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. Gastap: A gas analyzer for smart contracts. *arXiv preprint arXiv:1811.10403*, 2018.

[6] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017.

[7] J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*. Cambridge University Press, New York, NY, USA, 1st edition, 2009. ISBN 0521820499, 9780521820493.

[8] Santiago Bragagnolo, Henrique Rocha, Marcus Denker, and Stéphane Ducasse. Smartinspect: solidity smart contract inspector. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–18. IEEE, 2018.

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

[10] 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 (SANER)*, pages 442–446. IEEE, 2017.

[11] 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 (ICSE-NIER)*, pages 81–84. IEEE, 2018.

[12] Way L. Choy. When smart contracts are outsmarted: The parity wallet "freeze" and software liability in the internet of value - lexology. `https://www.lexology.com/library/detail.aspx?g=33a0af51-80f3-4643-8536-db9a0d9ba2c7`. (Accessed on 09/16/2019).

[13] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[14] Nigel Cutland. *Computability: An introduction to recursive function theory*. Cambridge university press, 1980.

[15] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, pages 79–94. Springer, 2016.

[16] Matteo Di Pirro. How solid is solidity? an in-dept study of solidity's type safety. 2018.

[17] Edsger W Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3): 147–148, 1968.

[18] Ardit Dika. Ethereum smart contracts: Security vulnerabilities and security tools. Master's thesis, NTNU, 2017.

[19] Lotte Fekkes, Lejla Batina, Louiza Papachristodoulou, and Joeri de Ruiter. Comparing bitcoin and ethereum. *URl: https://www. cs. ru. nl/bachelorscripties/2018/Lotte_Fekkes___ 4496426___Comparing_Bitcoin_and_Ethereum. pdf*, 2018.

[20] Maribel Fernández. *Programming languages and operational semantics*. Springer, 2014.

[21] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):116, 2018.

[22] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. Foundations and tools for the static analysis of ethereum smart contracts. In *International Conference on Computer Aided Verification*, pages 51–78. Springer, 2018.

[23] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 243–269. Springer, 2018.

[24] Robert Harper. Programming languages: Theory and practice. *Course Notes on the WWW*, 2002.

[25] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.

[26] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017.

[27] Marc Jansen, Farouk Hdhili, Ramy Gouiaa, and Ziyaad Qasem. Do smart contract languages need to be turing complete? In *International Congress on Blockchain and Applications*, pages 19–26. Springer, 2019.

[28] Hyesook Lim, Changhoon Yim, and Earl Jr. Priority tries for ip address lookup. *Computers, IEEE Transactions on*, 59:784 – 794, 07 2010. doi: 10.1109/TC.2010.38.

[29] Fuchen Ma, Ying Fu, Meng Ren, Wanting Sun, Zhe Liu, Yu Jiang, Jun Sun, and Jiaguang Sun. Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability. *arXiv preprint arXiv:1910.02945*, 2019.

[30] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *International Conference on Financial Cryptography and Data Security*, pages 523–540. Springer, 2018.

[31] Bernhard Mueller. Laser-ethereum: Symbolic virtual machine for ethereum. `https://github.com/b-mueller/laser-ethereum`. (Accessed on 09/16/2019).

[32] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. In *9th Annual HITB Security Conference (HITBSecConf)*, 2018.

[33] Jerzy Mycka, Francisco Coelho, and José Félix Costa. The euclid abstract machine: trisection of the angle and the halting problem. In *International Conference on Unconventional Computation*, pages 195–206. Springer, 2006.

[34] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[35] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 1846286913.

[36] JaeYong Park, Daegeon Lee, and Hoh Peter In. Saving deployment costs of smart contracts by eliminating gas-wasteful patterns. *International Journal of Grid and Distributed Computing*, 10(12): 53–64, 2017.

[37] Carlos Pérez Jiménez. Analysis of the ethereum state.

[38] John C Shepherdson and Howard E Sturgis. Computability of recursive functions. *Journal of the ACM (JACM)*, 10(2):217–255, 1963.

[39] Matt Suiche. Porosity: A decompiler for blockchain-based smart contracts bytecode. *DEF con*, 25: 11, 2017.

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

[41] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[42] Patrick Ventuzelo. Powerpoint presentation. `https://patrickventuzelo.com/wp-content/uploads/2018/10/hack_lu_2018_Reversing_and_Vulnerability_research_of_Ethereum_Smart_Contracts.pdf`. (Accessed on 09/17/2019).

[43] Shuai Wang, Yong Yuan, Xiao Wang, Juanjuan Li, Rui Qin, and Fei-Yue Wang. An overview of smart contract: architecture, applications, and future trends. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 108–113. IEEE, 2018.

[44] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[45] Beatriz Xavier. Formal analysis and gas estimation for ethereum smart contracts. Master's thesis, Instituto Superior Técnico, Departamento de Matemática, 2018.