Formal Analysis of Ethereum Virtual Machine Bytecode Patterns

Bernardo Gastão Varanda da Silva Prates
bernardo.prates@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

December 2019

Abstract

Ethereum became noticeable for implementing computerized protocols, smart contracts. The transaction based system that it constitutes is capable of executing these programs in a deterministic environment and to keep the registers associated with these actions available publicly.

In order to prevent the system against abuses such as the Denial-of-Service attacks, computational work is measured in terms of Gas. Ether, the cryptocurrency embedded in the system, is used to purchase the amount of Gas 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 the alternative patterns constitute indeed an optimization.

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 and determine if optimization may be applied.

Although the Ethereum Virtual Machine (EVM) is not Turing Complete, characterizing a contracts behavior is still a difficult task. We prove that with few modifications, mainly in terms of storage capacity augmentation, 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, Blockchain, Smart Contracts, Operational Semantics, Turing Completeness

1. Introduction

Ethereum is the main platform for smart contracts [9]. Comparing with Bitcoin, which also relies on the Blockchain technology and is equipped with a scripting language, the range of contracts that are allowed to be written is substantially wider.

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.

In order to perform a transaction, the sender must pay a fee upfront in Ether. 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, $\$1359.48$ (January, 14, 2018, according to [1]). In accordance, the safety and security of smart contracts is a trendy subject. Although the hardships associated to the task of scrutinizing the low-level bytecode programs the contracts are compiled to may pose a barrier to a thorough analysis of smart contracts.

2. Ethereum Overview and Related Work

Labeled as the ”World Computer”, Ethereum consists of a platform for performing deterministic computations in a decentralized fashion.

Storage in Ethereum is accomplished by the use of a data structure named Merkle Patricia tree.

Gathering all the information generated by this system is achieved by four types of databases: State Trie; Storage Trie; Transaction Trie; Receipts Trie.

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.

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

The payment for transactions is performed prior to execution. A mechanism to measure the computational power required for execution is managed the element gas.

Blocks are structures containing a collection of transactions, an identification number and a nonce, a numerical value necessary for block validation.

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.

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

The EVM operates on five memory addresses: the code, the input data, the stack, the memory and the account storage.

The execution stack pertains to the last in first out category, and stores 256-bit words, with no cost associated. Although the maximum capacity of the stack is fixed at 1024 elements, only the first 17 can be accessed directly.

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.

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 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. Similarly to memory, at first, all entries are set to zero.

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.

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 [2].

A first semantic definition of the EVM was elaborated by the Ethereum co-founder Gavin Wood in the Ethereum Yellow Paper [10].

3. EVM Semantics and Optimization

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.* [4], corrected and completed by Xavier in [11], 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. Syntax

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 $\mathcal{A} = \{(n, b, stor, code) |\ n, b \in \mathbb{N}_{256},\ stor \in \mathbb{B}^{256} \rightarrow \mathbb{B}^{256},\ code \in (\mathbb{B}^8)^*\}$.

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 account -* 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.

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.*

3.1.1 Execution States

An element of the execution state which is already introduced in the Ethereum Yellow Paper [10] 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.

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 $(\mu, \iota, \sigma, \eta) :: S \in \mathbb{S}$.

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 $\mathbb{S} = M \times I \times \Sigma \times N$*.*

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.

3.2. 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.* [4] with the purpose of identifying

the current opcode in execution:

$$\omega_{\mu,\iota} = \begin{cases} \iota.\mathsf{code}[\mu.\mathsf{pc}] & \mu.\mathsf{pc} < |\iota.\mathsf{code}| \\ \mathsf{STOP} & otherwise \end{cases} \tag{1}$$

Equation 1: 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.

$$\frac{\omega_{\mu,\iota} = \mathsf{OP} \quad premises(S)}{\Gamma \vDash S \to S'} \qquad S, S' \in \mathbb{S},\ \Gamma \in \mathbb{T}_{env} \tag{2}$$

Equation 2: Small-Step Rule Structure

An important feature concerning the small-step relation $\to$ 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 $\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to S'$, for $S' \in \mathbb{S}$.

The beginning and completion of transactions is abstracted by two auxiliary functions: $initialize(\cdot,\cdot,\cdot) \in \mathbb{T} \times \mathcal{B} \times \Sigma \to (\mathbb{T}_{env} \times \mathbb{S}) \cup \{\bot\}$ 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:

$$\frac{(\Gamma, S) = initialize(T, B, \sigma) \quad \Gamma \vDash S \to^* S' \quad final(S') \quad \sigma' = finalize(T, \eta', S')}{\sigma \xrightarrow{T,B} \sigma'} \tag{3}$$

Equation 3 Transaction formal display as in [4]

3.3. Operational Semantics

An action is given by an opcode and the respective sequence of arguments. The set of actions is denoted by Act.

The small-step relation portrays transitions between execution states corresponding to call stacks, $\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to S'$, 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 $S'' \in \mathbb{S}$

$$\frac{}{\Gamma \vDash S \to^* S} \qquad \frac{\Gamma \vDash S \to^* S' \quad \Gamma \vDash S' \to S''}{\Gamma \vDash S \to^* S''} \tag{4}$$

Equation 4 Evaluation Relation Rules

4. 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:* $\mu''.\mathsf{gas} > \mu'.\mathsf{gas}$, $\iota'.\mathsf{code} \neq \iota''.\mathsf{code}$, $\sigma'(a).\mathsf{code} \neq \sigma''(a).\mathsf{code}$ *(*a *standing for the contract address* $\iota.\mathsf{actor}$*), and* $\mu'.\mathsf{pc}$ *may differ from* $\mu''.\mathsf{pc}$.

Additionally, the actions to be optimized, must not produce an exceptional state for all contexts, i.e.,
$\exists \Gamma \in \mathbb{T}_{env},\ \mathcal{C}[\cdot] = (\mu, \iota, \sigma, \eta) :: S,$ *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.*

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, 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 [11].

Proposition 2.

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

$n = 1, \ldots, 16$

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 $\pi = \mathsf{SWAP}n\ \mathsf{SWAP}n,\ n = 1 \ldots 16,\ \Gamma \vDash \mathcal{C}[\pi] \xrightarrow{\pi}{}^{*} (\mu', \iota', \sigma, \eta) :: S$, 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 $\mathsf{SWAP}n$, the consecutive execution of the same instruction $\mathsf{SWAP}n$ 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).

$$\mu'.\mathsf{s} = s_0 :: \ldots :: s_n :: s = \mu.\mathsf{s}$$
$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 6$$
$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 2$$

□

Proposition 3.

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

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

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:

$$\mu'.\mathsf{s} = s_0 :: s = \mu.\mathsf{s},$$
$$\mu'.\mathsf{gas} = \mu.\mathsf{gas} - 6$$
$$\mu'.\mathsf{pc} = \mu.\mathsf{pc} + 2$$

□

5. Turing Completeness for a Modified EVM and Control Flow Decision Problems

The notion of computability is intrinsically related to the respective computational model. 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 further on, was generally specified in order to fulfill the requirements of the proof.

5.1. Unlimited Register Machine

The representation of the Unlimited Register Machine presented by Cutland in [3] and the proof of Turing completeness from [8] will guide the various topics that will be discussed in this section.

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

$$R_1, R_2, R_3, R_4, \ldots$$

having as states space $\Omega = \{\omega \mid \omega$ *is 0 almost everywhere*$\}$ *- and each state consists of a sequence* $\omega = (\omega_1, \omega_2, \omega_3, \omega_4, \ldots)$*, 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* $P = I_1, I_2, \ldots, I_s$ *is a sequence of instructions belonging to one of the four categories:*

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

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

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

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

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

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

- *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* $a_1, a_2, a_3, \ldots$*, which are placed in the respective registers* $R_1, R_2, R_3, \ldots$*, 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 $P = I_1, I_2, \ldots, I_s$.

Definition 17. *Let* $a_1, a_2, a_3, \ldots$ *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* $n > 0$ *an* $a_1, a_2, \ldots, a_n, b \in \mathbb{N}$*, 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* $J(m, n, s_1 + q)$*.*

5.2. 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 [5]. 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.

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 that 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 $(n = 1, \ldots, 32)$ 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 $w = b_1 b_2 \ldots b_n$ with n bytes is performed by the variable length encoding scheme h given below:

$$h(w) = b'_1 b_1 b'_2 b_2 \ldots b'_n b_n,$$

  where $b'_i = \texttt{0x01}$, if $i \neq n$, and $b'_i = \texttt{0x00}$, 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.

5.3. 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 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 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 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: $J(m, n, q)$ 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 4. *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 $Z(m)$). □

5.4. 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 [3].

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 $f_P^{(n)}$.

5.4.1 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.*

Theorem 1. *The sets $\mathscr{I}$, $\mathscr{P}$ are effectively denumerable and the set $\mathscr{C}$ is denumerable.*

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 (encoded by the number 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.*

5.4.2 Undecidability of Modified EVM Decision Problems

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 [3], a perspective that this work will follow.

Proposition 5. *The problem of, given $x \in \mathbb{N}$, whether $x \in W_x$ is undecidable.*

Proposition 6. *Given $x, y \in \mathbb{N}$, the problem of determining whether $\phi_x(y)$ is defined (or equivalently $P_x(y) \downarrow$) is undecidable.*

Proposition 7. *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 2. *Given a modified EVM program $M = e_1, e_2, \ldots, e_n$, such that there exists at least one occurrence of $e_i =$ 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 $P = I_1, I_2, \ldots, I_s$ in the standard form, it is possible to translate P to the modified EVM bytecode (the resulting program shall be named M_P).

By appending to the code of M_P the instructions:

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

where k′ corresponds to the number of bytes of b and k is the smallest positive integer to $s + 10 + 2k + 2k'$.

As a consequence, by applying algorithm J, one would be able to affirm for the program P if the JUMPI eventually executes the jump to the address $\mathtt{S} + 9 + 2\mathtt{k} + 2\mathtt{k}'$, thus, also revealing whether b belongs in E_P. □

Theorem 3. *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 6, 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 $P' = I'_1, I'_2, \ldots, I'_s$ in the standard form, it is possible to translate P' to the modified EVM bytecode (the resulting program shall be named $M_{P'}$).

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

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

where k the smallest positive integer corresponding to the number of bytes 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. □

6. Smart Contract Optimizer

6.1. Smart Contract Structure

A compiled contract consists of three components: the loader code, the runtime code and the swarm hash.

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.

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 [11].

Definition 24. *A bytecode block b_j of a program $P = i_0, i_1, \ldots, i_n$ is denoted by a sequence of instructions $b_j = b_{j,0} \ldots b_{j,k}$ such that:*

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

Definition 25. *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* $a \notin \{\text{JUMP}, \text{STOP}, \text{RETURN}, \text{REVERT}, \text{INVALID}\}$*.*

6.2. 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.

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 [7].

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 [6].

7. 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:

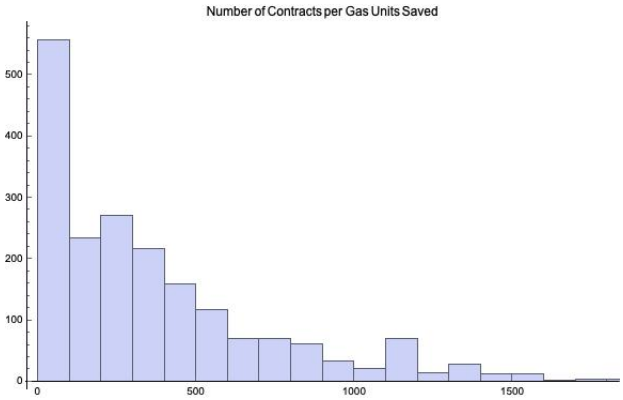

Figure 5: Number of contracts per gas saved from the optimization experiment

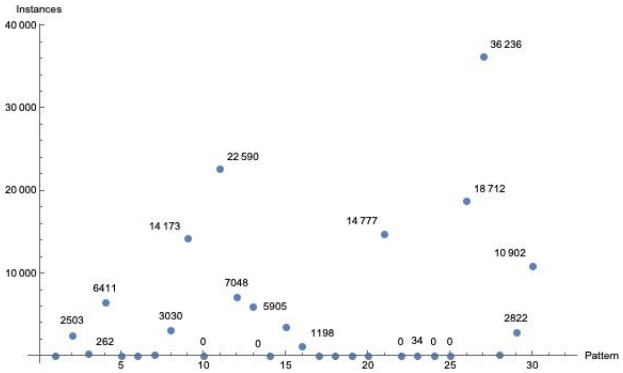

Figure 6: Number of instances of each bytecode pattern found throughout the optimization experiment

8. Conclusions

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 jump instructions are unaffected. Nevertheless, 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.

References

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

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

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

[4] I. Grishchenko, M. Maffei, and C. 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.

[5] M. Jansen, F. Hdhili, R. Gouiaa, and Z. Qasem. Do smart contract languages need to be turing complete? In *International Congress on Blockchain and Applications*, pages 19–26. Springer, 2019.

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

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

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

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

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

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