Formal analysis and gas estimation for Ethereum smart contracts

Beatriz Henriques Xavier
Instituto Superior Técnico, Lisboa, Portugal

December 2018

Abstract

Ethereum provides an ecosystem for the development of smart contracts – programs stored in a public blockchain that may interact with each other. Formal verification systems for these platforms are of great importance since the modifications made on the blockchain are irreversible and may concern tokens with monetary value. We study the Ethereum Virtual Machine and implement a symbolic Ethereum Virtual Machine in Mathematica that is able to run smart contracts with symbolic input parameters. This machine is used in the development of a formal analysis tool that symbolically executes every execution path of a smart contract, returning all possible final states of the system. This exhaustive approach allows a user to verify properties about the global state of the system after the execution of a given contract. Moreover, we extend the semantics of the classic virtual machine to this symbolic virtual machine. Executing code on the blockchain has a cost that depends both on the global state of the system and on the input given to the code. We explore the topic of gas costs and use the symbolic virtual machine to build a tool that manages to estimate the cost of executing a function of a contract that extends previously existing tools, since it not only considers a broader class of contracts, but also returns more detailed information. Namely, the tool accepts code containing loops with a fixed number of iterations and returns a cost for every execution path, rather than a single upper estimate.

Keywords: formal verification, symbolic execution, control flow analysis, operational semantics, Ethereum, blockchain

1 Introduction

Ethereum is a platform designed to serve as a ledger of financial transactions and decentralized applications. Inspired by the Bitcoin, launched in 2009 [8], Vitalik Buterin created and first described Ethereum in early 2014 in a white paper [1]. Bitcoin is a cryptocurrency system functioning as a bank in a public, decentralized database called blockchain. Transactions are validated by some users, the miners, through cryptographic algorithms [3] that verify their legitimacy, and are stored in blocks. Thus, the blockchain, an ordered sequence of blocks, is built collectively by the users and the accepted state of the blockchain at any moment is determined by the majority of the miners.

The main idea behind the development of Ethereum was to expand the concept of blockchain to other applications besides monetary transactions, such as proving property ownership, establishing decentralized autonomous organizations, voting, gambling, or attributing automatic insurance compensations. To achieve this end, Ethereum developed a virtual machine that specifies a *quasi*-Turing-complete language over which these applications, called smart contracts [11], must be written.

Every computation in the Ethereum framework is bounded by *gas*. The network has its own currency, ether, given as a reward to the miners that contribute to the maintenance of the blockchain. The full description of the Ethereum environment and of the Ethereum Virtual Machine (EVM) was published in a yellow paper by the co-founder Gavin Wood [14].

Given that Ethereum wishes to be a project as open and global as possible, it also contains high-level languages that are compiled to EVM bytecode, the most popular of which is Solidity. One major security concern arises precisely from the fact that it is really easy to program a smart contract. Although smart contracts are usually written in diverse high-level languages, every piece of code is compiled to the common core of the Ethereum network that is EVM bytecode. For that reason, we have decided that a complete analysis of smart contracts should be performed at the bytecode level. In a framework as plural as Ethereum, that combines currency and functionality, that was designed to be used at a global scale, and that already suffered some attacks, a formal verification system that provides the developers a better understanding about the behaviour of their contracts

is of the uttermost importance.

In this work, developed in collaboration with Tekever, we have designed and implemented in Mathematica an abstraction of the Ethereum Virtual Machine that is able to symbolically execute interactions between given smart contracts and return all possible final states of the system – resultant from all possible execution paths – and the conditions under which each path is taken.

The main contributions of this work are the following:

- Implementation of a symbolic Ethereum Virtual Machine in Mathematica that allows the execution of a function of a smart contract with symbolic input and considering symbolic account contents, retrieving a comprehensive list of the possible final states;
- Adaptation of the small-step semantic rules of the classic virtual machine to the symbolic virtual machine;
- Development of a gas estimator for smart contracts that contain loops with a fixed number of iterations.

The present article is organized as follows. Chapter 2 introduces the Ethereum framework, defining the components that make up the Ethereum environment – which includes the virtual machine architecture and operations, as well as its relationship with the Ethereum environment. In Chapter 3 we examine the symbolic model, starting with a general description of the functionalities and motivation for some needed design decisions and continuing by specifying the semantic rules that govern the symbolic machine, finishing with the analysis of a real smart contract that was exploited. Chapter 4 explores a different subject: it concerns estimation of gas costs of executing smart contracts. A tool that estimates the cost of executing a function of a contract is developed and explored through a real example. The conclusions of this work are summarized and future research directions are suggested in Chapter 5.

2 The Ethereum Virtual Machine

This chapter introduces the Ethereum blockchain and the Ethereum Virtual Machine in more detail. Accounts are a central concept of Ethereum. An account can store, send, and receive ether, similarly to a bank account. However, Ethereum accounts have more functionalities: internal storage and executable code. A *smart contract* is simply an account that has associated code, as we will see in the following subsection. The Ethereum Virtual Machine provides a programming language (a finite set of operations) and an environment for the programs to run constituted by elements such as a volatile memory and an execution stack.

The EVM is a stack-based machine working on arithmetic modulo 2^{256}. Its elements are, thus, 256-bit numbers. Its operations, also known as opcodes, form a Turing-complete language, but the machine has an additional feature: the amount of computational work done in every execution is bounded by gas. Gas has a cost in ether and the gas given to the execution of a transaction is specified by the user that started it. Every operation has a positive gas cost, so it must stop. If it stops due to lack of gas, all changes made are reverted but the user that triggered the execution is charged. The following subsection establishes the formal definitions of the basic concepts of Ethereum. Ethereum uses the standard cryptographic hash functions `KECCAK-256` and `KECCAK-512` for the generation of account addresses, signatures of transactions, validation of blocks, among others. In the present work, the expression *hash function* should be read as *cryptographic hash function*.

2.1 Basic definitions

Let $\mathbb{A} = \{0,1\}^{160}$ be the set of all possible addresses, $\mathbb{B} = 0, ..., 255$ the set of bytes and $\mathbb{N}_{256}$ the set of natural numbers below 2^{256}. S^* denotes the Kleene closure of the set S and ε denotes the symbol such that $\{\varepsilon\}$ is the empty element of S^*.

The formal definition of *account* is relative to account contents; the link between addresses and account contents is a part of the global state of the system.

Definition 1 (Account)**.** *An account is a tuple* $A = (b, code, stor, n)$*, where:*

- $b \in \mathbb{N}_{256}$ *is the balance;*
- $code \in \mathbb{B}^*$ *is the code. The account is a contract if* $code \neq \varepsilon$ *and externally owned otherwise;*
- *stor is a function from* $\{0,1\}^{256}$ *to* $\{0,1\}^{256}$ *and it is the storage;*
- $n \in \mathbb{N}_{256}$ *is the nonce. If the account is externally owned,* n *is the number of transactions sent from it. If the account is a contract,* n *is the number of account creations made by the account.*

Definition 2 (Global state)**.** *The global state of the system is a function between the set of addresses and*

the set of accounts:

$$\sigma \colon \mathbb{A} \to \mathbb{N}_{256} \times \mathbb{B}^* \times (\{0,1\}^{256} \to \{0,1\}^{256}) \times \mathbb{N}_{256}$$

We convention that the account with address *a exists* if $\sigma(a) \neq (0, \varepsilon, 0_F, 0)$, where 0_F is the constant function whose value is always zero. We convention that the empty account is represented by $\perp$. Transactions are the means of communication between accounts.

Definition 3 (Transaction)**.** *A transaction is a tuple* $T = (from, to, n, v, d, g, p)$*, where:*

- $from, to \in \mathbb{A}$ *are the sender and the receiver addresses of the transaction, respectively;*
- $n \in \mathbb{N}_{256}$ *is the nonce of the account with address* $from$*;*
- $v \in \mathbb{N}_{256}$ *is the value;*
- $d \in \mathbb{B}^*$ *is the data;*
- $g \in \mathbb{N}_{256}$ *is the gas limit: the maximum amount of gas that should be used in this transaction;*
- $p \in \mathbb{N}_{256}$ *is the gas price.*

If $to = 0$*, then* T *is an account creation transaction and* d *is code used to determine the code of the new account. Otherwise,* d *is the data given as input to the execution of the code of the account with address* to*.*

When a transaction is broadcasted to the blockchain, it will be executed by some users, called *miners*. Every transaction involves the payment of a fee in ether, equal to the gas spent times the gas price. The gas price is chosen by the sender and the gas spent is never greater than the gas limit, so the sender knows upfront the maximum fee that he may have to pay. If there is not enough gas, the transaction is declared invalid and its effects are not committed. If there is enough gas, the transaction is completed and the miners start trying to solve the proof of work. The first miner that completes it broadcasts that information and the other miners check that he did it. Then, every miner executes the transaction and updates its own database to reflect the changes that were made to the system.

Transactions are recorded inside blocks, the fundamental unit of the blockchain. Blocks contain several components, including references to previous blocks and to data structures that store information about the transactions of the block.

Definition 4 (Block)**.** *A block is a tuple* $B = (nonce, T, i, l, ben, d)$ *where:*

- $nonce \in \mathbb{B}^8$ *is the nonce;*
- $\mathcal{T} = (T_1, \ldots, T_m)$ *is a list of transactions;*
- $i \in \mathbb{N}$ *is the number;*
- $l \in \mathbb{N}$ *is the gas limit;*
- $ben \in \mathbb{A}$ *is the address of the account of the beneficiary;*
- $d \in \mathbb{N}$ *is the difficulty.*

The blockchain is, therefore, an ordered sequence of blocks.

2.2 Ethereum Virtual Machine overview

The architecture of the EVM is simple: it is a stack machine that operates on 256-bit elements. Its operations, also known as opcodes and represented by numbers from 0 to 255, contain functionalities from simple arithmetic operations to account creation and deletion. The stack has capacity for 1024 elements; if it overflows, an exception is raised and all changes made during the execution are lost. A *program* is a finite sequence of bytes – if a byte does not represent any existing opcode, we say that it is an *invalid* opcode.

An execution of a program has a volatile memory associated with it, defined as an infinite byte array that starts empty and that is cleared when the execution finishes. There is an operation, `MSTORE`, that writes an element of the stack in a specified position the memory. The difference between the designs of the stack and of the memory causes the writing operation to split the element to store in 32 bytes and to write each byte in a contiguous position of the memory, starting from the specified position. It is also possible to access 32 contiguous bytes of the memory using the operation `MLOAD`. The storage of an account can be accessed and modified by the virtual machine – opcodes `SLOAD` and `SSTORE`, respectively – but, unlike what happens with the memory, the changes made to the storage are permanent. It is normally used to write the values of global variables of a program: for instance, if a smart contract implements a bank, the amount that each client owns may be updated by functions such as *withdraw* or *deposit* and must not be lost between executions.

Since the word size of the EVM is 256 bits, all arithmetic operations are modulo 2^{256} – like `ADD`, `MUL`, `SUB`, `DIV`, or `EXP`. There are several comparison operators that return 1 or 0 if the comparison returns true or false, respectively: `LT`, `GT`, `EQ`, bitwise `AND`, bitwise

OR, bitwise XOR and bitwise NOT are some of them. It is also possible to PUSH an element with $n \leq 32$ bytes to the stack, to POP the top element, to duplicate an existing element up to the 16th position of the stack (DUP), or to SWAP the first element with other member of the stack, up to the 17th. Finishing an execution with or without output corresponds to executing RETURN or STOP. The cryptographic hash function KECCAK-256 is embedded in the virtual machine through the opcode SHA3. The virtual machine also provides access to the environment where the program is executed: some opcodes concern block information, some are relative to the transaction, and others access information about other accounts.

The most distinctive and important operations of the EVM are related to creation, interaction and destruction of accounts. Account creation is achieved using CREATE. This operation generates a new address, transfers the specified value to the new account, and causes the current execution to pause, starting the execution of the code d that contains instructions to build the code of the new account in a fresh execution stack with a fresh memory. When this execution finishes, the original one continues. It is also possible to destroy an account through the opcode SELFDESTRUCT. Finally, a transaction from an account A to an account B may be performed by three opcodes: CALL, CALLCODE, or DELEGATECALL. The first action is the transference of the designated value to B. If B is an externally owned account, the transaction finishes. Otherwise, similarly to what happens during account creation, the code of B runs with the desired input in a new stack, with a new memory and considering the environment of the execution of B. The execution of A waits for the execution of B to finish before resuming.

Conditional execution of code is possible due to the opcode JUMPI, and unconditional execution is achieved through JUMP. Both opcodes receive as an argument the position of the code where the program counter should jump to, and JUMPI receives additionally a boolean that flags whether the jump should be performed.

2.3 Execution model

The execution model intends to formalize the Ethereum Virtual Machine components as well as its interaction with the environment. Recall that a contract A may call other contract B, triggering its execution. It is important to distinguish between elements that are relative to the execution of the code of A and elements that are relative to the entire transaction that starts with the execution of A, including instructions of other contracts that were called by A, such as B.

$$\begin{aligned}
\mu &= (gas, pc, m, i, s, rd) \\
&\in \mathbb{N}_{256} \times \mathbb{N}_{256} \times \mathbb{B}^* \times \mathbb{N}_{256} \times (\{0,1\}^{256})^{1024} \times \mathbb{B}^* \\
\iota &= (actor, input, sender, value, code) \\
&\in \mathbb{A} \times \mathbb{B}^* \times \mathbb{A} \times \mathbb{N}_{256} \times \mathbb{B}^* \\
\eta &= (bal_r, L, S_\dagger) \\
&\in \mathbb{N}_{256} \times \mathcal{L} \times \mathcal{P}(\mathbb{A}) \\
\Gamma &= (o, price, B) \\
&\in \mathbb{A} \times \mathbb{N}_{256} \times \mathcal{B}
\end{aligned}$$

Figure 1: Machine state, execution environment, transaction effect and transaction environment components

The *machine state* μ and the *execution environment* ι concern the execution of a single contract. While the former models the elements whose values change during the execution – such as available *gas*, memory m, execution stack s, or return data rd (not present in the original definition [4] and added during the present work) –, the latter contains the fields that are constant – possible examples are the currently executing contract, *actor*, or the contract that started the current execution, *sender*. The *transaction effect* η includes information about the entire transaction that is updated during the execution, but that only has effect when it finishes – like the balance refunds that the miner should receive, bal_r –, and the *transaction environment* Γ is the list of parameters that are constant during the transaction execution – such as the account that started the present sequence of transactions, o, or the block where the transaction is being executed, B. Their components and domains are summarized below, where $\mathcal{B}$ and $\mathcal{L}$ denote, respectively, the set of blocks and the set of logs, whose structure is rather complex, and $\mathcal{P}(\mathbb{A})$ represents the class of sets of addresses. These structures are represented in Figure 1.

A transaction is modelled using a *call stack*. An element of the call stack models the execution of a contract and it is of the form $(\mu, \iota, \sigma, \eta)$ – where μ, ι, and η are as above and σ is the global state –, or EXC, or $HALT(\sigma, g, d, \eta)$ – where $g \in \mathbb{N}_{256}$ is the remaining gas and $d \in \mathbb{B}^*$ is the output data –, or $REV(g, d)$. EXC signals that an exception occurred, $HALT$ represents successful termination and REV, introduced by this work and motivated by the release of the opcode REVERT, models unsuccessful termination where the caller does not lose all gas sent to the

$$
\begin{aligned}
S :=& HALT(\sigma, g, d, \eta) :: S_{plain} \mid EXC :: S_{plain} \\
& REV(g, d) :: S_{plain} \mid S_{plain} \\
S_{plain} :=& (\mu, \iota, \sigma, \eta) :: S_{plain}
\end{aligned}
$$

Figure 2: Grammar for a call stack S

$$
\frac{\omega_{\mu,\iota} = \texttt{OP} \quad premises(S)}{\Gamma \models S \to S'}
$$

Figure 3: General structure of a small-step rule in the symbolic machine

execution. The grammar for the call stacks is summarized in Figure 2.

A transaction is initiated by an external message: an action of an externally owned account. The transaction $T = (from, to, n, v, d, g, p)$ and the current global state σ contain all the information needed to initialise the call stack. The initial call stack contains a single element $(\mu, \iota, \sigma, \eta)$ such that:

- $\mu = (g, 0, \varepsilon, 0, \varepsilon, \varepsilon)$;
- $\iota = (to, d, from, v, \sigma(to).code)$;
- σ is the global state function in the beginning of the execution;
- $\eta = (0, \emptyset, \emptyset)$.

The small-step rules that govern the EVM operations, as is usual in operational semantics ([5],[9]), contain the premises at the top and the consequences at the bottom, as Figure 3 shows, where $\omega_{\mu,\iota}$ denotes the current opcode and S, S' are call stacks.

3 Formal verification of EVM bytecode

The main contribution of this work is the development of a symbolic Ethereum Virtual Machine. In this section we formally define the components of the symbolic machine, presenting its features and limitations. A small-step semantic rule is explored in order to illustrate the scope of the model. Finally, we examine a real smart contract and are able to detect a vulnerability in it that was previously known.

3.1 Symbolic model

Since the goal is to consider every possible outcome of an execution, the state of the execution of a transaction cannot be modelled by a call stack anymore: it is now a list of call stacks, each representing a possible execution path.

We analyse a contract assuming that we know the contracts that it may call or that may call it. Therefore, we state that a given property is valid or invalid in a fixed context. A valid property in an environment may be violated by an interaction with a contract from outside that environment. This assumption, although strong, is hard to overcome since any contract may call other contract sending an arbitrary byte array as input so, in general, the result is unpredictable.

The model has some limitations: not every component of the machine state, global state, execution environment, transaction effect and transaction environment is allowed to be symbolic. We will consider an infinite, enumerable set $\mathcal{S} \supseteq \mathbb{N}_{256}$ of symbols and will denote in **bold** the elements that cannot be instantiated with a symbol that is not an element of $\mathbb{N}_{256}$.

3.2 Semantics of the symbolic EVM

When the execution reaches a point such that the next step depends on a symbolic, unknown value, a fork occurs and all possibilities for the unknown value are considered. Execution paths are then identified by formulas that contain the assumptions on the symbolic variables made during the execution of the path. We will use notations for first-order logic from [10]. Hence, the machine state μ of the symbolic machine contains an additional field, $cond \in L_\Sigma(\mathcal{S})$, where $\Sigma = (\mathcal{F}, \mathcal{P}, \tau)$ is a signature containing the following function ($\mathcal{F}$) and predicate ($\mathcal{P}$) symbols:

$$
\begin{aligned}
\mathcal{F} =& \{+, -, \times, \div, \lfloor\rfloor, exp, mod, TC, \&, ||, \oplus, sdiv, \\
& smod, byte\} \\
\mathcal{P} =& \{<, \leq, >, \geq, =, \neq, isAddress, isBool\}
\end{aligned} \tag{1}
$$

τ is the expected arity function and formulas are built from the set of terms $T_\Sigma(\mathcal{S})$ from the predicate symbols and the symbol $\wedge$, since, usually, a modification to the condition consists of a conjunction of a new term.

The EVM components that can be symbolic (or, in the case of lists or arrays, the components that can contain symbolic elements) are represented in **bold**:

- $\mu = (gas, pc, \mathbf{m}, i, \mathbf{s}, \mathbf{rd})$;
- $\iota = (\mathbf{actor}, \mathbf{input}, \mathbf{sender}, \mathbf{value}, code)$;
- $\eta = (bal_r, \mathbf{L}, \mathbf{S}_\dagger)$;
- $\Gamma = (\mathbf{o}, \mathbf{price}, \mathbf{B})$.

In what concerns account contents, addresses may be represented by symbols, as well as balances and storage contents. Values of transactions can be symbolic, which is of great use for property verification since often one is interested in knowing if the execution of a certain piece of code may cause him to lose ether. Memory, input and return data are arrays that can handle symbols in a special way. Since the word size of the EVM is 256 bits, or 32 bytes, and many operations concerning memory, input, or return data handle blocks of 32 bytes instead of single bytes, we have decided that a symbolic element of one of these fields would be represented by 32 bytes containing the same symbol. This means that storing a symbol $\mathbf{x}$ in the position p of the memory is represented by storing an $\mathbf{x}$ in every position from p to $p+31$; and that asking for the word of the memory that starts at position p results in the outcome $\mathbf{x}$.

The general structure of a rule changes to encompass multiple call stacks. Given that the universe is now a list of call stacks, a step consists of the execution of an instruction in each call stack, namely, the instruction in the position that corresponds to the program counter of the machine in each of the top elements of the call stacks. We will write the list of call stacks vertically for better readability. If there are n call stacks, the evolution of the call stack $(\mu_i, \iota_i, \sigma_i, \eta_i) :: S_i$ is described by the rule that is applicable to the opcode $\texttt{OP}_i$ in the conditions $premises(S_i)$. The general structure is simple to derive from Figure 3.

Since the executions of the n call stacks are independent from each other, we only write one call stack on the left hand side and the call stacks that result from that one on the right hand side. We say that we reach a final state when there is no rule that can be applied to any of the call stacks $S_1, \ldots, S_n$.; this way we are looking for a fixed point of the function that applies the small-step rules.

Initialisation is very similar to the numeric case: the only modification is in the initial value of the machine state μ: it is now $(g, 0, \varepsilon, 0, \varepsilon, \varepsilon, \mathbf{cond})$, where $\mathbf{cond} \in L_\Sigma(\mathcal{S})$ is the initial condition that we wish to specify, usually related to the symbolic variables that are present in the initial global state σ. The finalisation should contain the payment of rewards to the miners and the destruction of the accounts in the suicide set.

The semantic rules of the EVM [4] are slightly changed in order to reflect the characteristics of the symbolic model. As an example, we present in Figure 4 one of the rules that models the very important operation `CALL`; it concerns the case where an existing account is called. Notice that the input $\mathbf{d}$ may contain symbols. Although we will not detail input handling in the symbolic machine in the present abstract, it should be mentioned that the first four bytes of it must be numeric, since usually that represents the function of the contract $\mathbf{to}_a$ that should be called; and that each of the next blocks of 32 bytes, which represent the arguments of the function, must be either numeric or contain 32 copies of the same symbol. This instruction pauses the current execution and starts the execution of the called contract, which is modelled by pushing a new element $(\mu, \iota, \sigma, \eta)$ to the call stack, which will be popped once the inner execution finishes so that the outer execution can resume.

3.3 Vulnerability detection in a real contract

We will illustrate the functionalities of the symbolic Ethereum Virtual Machine using a real contract that was discovered to contain a vulnerability in April 2018. This contract implements a coin, or token, named BecToken. We will focus on the function `batchTransfer` that is used to make a simultaneous transference of a given value to multiple addresses. The simplified code of this function can be seen in Figure 5. The functions `add` and `sub` are inherited from a standard library and implement, respectively, addition and subtraction modulo 2^{256}.

For the analysis of the contract we will assume that the array `rec` contains two elements $\mathbf{r_1}, \mathbf{r_2}$ that represent addresses of accounts whose initial balances are, respectively, the symbols $\mathbf{bal_1}, \mathbf{bal_2}$. The value of the parameter `value` will be denoted by the symbol $\mathbf{v}$. Moreover, the account that triggered the transaction is represented by **symbSender** and its initial balance is $\mathbf{bal_S}$. If the final balances of these accounts (after the execution of the function) are represented with a prime, one interesting property to verify would be the following:

$$\mathbf{bal_S} + \mathbf{bal_1} + \mathbf{bal_2} \stackrel{?}{=} \mathbf{bal'_S} + \mathbf{bal'_1} + \mathbf{bal'_2} \qquad (2)$$

The symbolic machine returned 42 final configurations, of which 27 correspond to successful termination and the remaining 15 to exceptional behaviour. The global states in the successful cases can be reduced to two classes. In the first one, the property held. In the second one, the final balances could be written as:

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \texttt{CALL} \qquad \mu.\mathbf{s} = g :: \mathbf{to} :: \mathbf{va} :: io :: is :: oo :: os :: \mathbf{s} \qquad |S| + 1 \leq 1024 \\ \mathbf{to}_a = \mathbf{to} \mod 2^{160} \qquad \sigma(\mathbf{to}_a) \neq \bot \\ aw = M(M(\mu.\texttt{i}, io, is), oo, os) \qquad \mathbf{d} = \mu.\texttt{m}[io, io + is - 1] \\ c_{call_1} = C_{gascap}(0, 1, g, \mu.\texttt{gas}) \qquad c_{call_2} = C_{gascap}(1, 1, g, \mu.\texttt{gas}) \\ c_1 = C_{base}(0, 1) + C_{mem}(\mu.\texttt{i}) + c_{call_1} \qquad c_2 = C_{base}(1, 1) + C_{mem}(\mu.\texttt{i}) + c_{call_2} \\ valid(\mu.\texttt{gas}, c_1, |\mathbf{s}| + 1) \qquad valid(\mu.\texttt{gas}, c_2, |\mathbf{s}| + 1) \\ \sigma' = \sigma\langle \mathbf{to}_a \to \sigma(\mathbf{to}_a)[\texttt{b} += \mathbf{va}]\rangle\langle \iota.\texttt{actor} \to \sigma(\iota.\texttt{actor})[\texttt{b} -= \mathbf{va}]\rangle \\ \mu'_1 = (c_{call_1}, 0, \lambda x.0, 0, \epsilon, \mu.\texttt{rd}, \mu.\texttt{cond} \wedge \mathbf{va} = 0) \\ \mu'_2 = (c_{call_2}, 0, \lambda x.0, 0, \epsilon, \mu.\texttt{rd}, \mu.\texttt{cond} \wedge 0 < \mathbf{va} \leq \sigma(\iota.\texttt{actor}).\texttt{b}) \\ \iota'_1 = \iota[\texttt{actor} \to \mathbf{to}_a][\texttt{input} \to \mathbf{d}][\texttt{sender} \to \iota.\texttt{actor}][\texttt{value} \to 0][\texttt{code} \to \sigma(\mathbf{to}_a).\texttt{code}] \\ \iota'_2 = \iota[\texttt{actor} \to \mathbf{to}_a][\texttt{input} \to \mathbf{d}][\texttt{sender} \to \iota.\texttt{actor}][\texttt{value} \to \mathbf{va}][\texttt{code} \to \sigma(\mathbf{to}_a).\texttt{code}]\end{array}}{\Gamma \models (\mu, \iota, \sigma, \eta) :: S \to \begin{array}{l}(\mu'_1, \iota'_1, \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ (\mu'_2, \iota'_2, \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S \\ EXC(\mu.\texttt{cond} \wedge \mathbf{va} > \sigma(\iota.\texttt{actor}).\texttt{b})) :: (\mu, \iota, \sigma, \eta) :: S\end{array}}$$

Figure 4: One of the rules for the successful execution of the `CALL` opcode in the symbolic machine

```
function batchTransfer(address[] rec,
uint256 value) public returns (bool) {
   uint cnt = rec.length;
   uint256 amt = uint256(cnt) * value;
   require(cnt > 0 && cnt <= 20);
   require(value > 0 && bals[msg.sender] >= amt);
   bals[msg.sender] = bals[msg.sender].sub(amt);
   for(uint i = 0; i < cnt; i++) {
      bals[rec[i]] = bals[rec[i]].add(value);
   }
   return true;
}
```

Figure 5: `batchTransfer` function code

$$\mathbf{bal'_S} = \mathbf{bal_S} - (2 \times \mathbf{v} \mod 2^{256}); \quad (3)$$

$$\mathbf{bal'_1} = (\mathbf{bal_1} + \mathbf{v}) \mod 2^{256}; \quad (4)$$

$$\mathbf{bal'_2} = (\mathbf{bal_2} + \mathbf{v}) \mod 2^{256}. \quad (5)$$

It is simple to verify that $\mathbf{v} = 2^{255}$ leads to $\mathbf{bal'_S} + \mathbf{bal'_1} + \mathbf{bal'_2} = \mathbf{bal_S} + \mathbf{bal_1} + \mathbf{bal_2} + 2^{256}$, which violates the property and states that it is possible to generate 2^{256} monetary units from nothing. The contract suffered precisely this attack, but it was quickly detected since there was a transaction involving an unusually high amount of tokens. However, the logical flaw that allowed the vulnerability to exist – arithmetic overflow – can be present in other contracts, as well as other flaws that have been not thought of yet. For this reason, we believe that a formal verification system is very important.

4 Estimation of gas costs

Program execution has a real cost for the entity that triggers it. The cost in ether of executing a contract depends on the gas price of the transaction and on the gas cost of the operations performed. Since gas prices are determined by the market and gas costs are fixed (for each operation, arguments and environment), estimating gas costs is very important for Ethereum blockchain users. The main aim of this section is to develop a systematic procedure to estimate the gas cost of calling a function of a contract as precisely as possible using the symbolic EVM. We will assume that the contract bytecode was compiled from Solidity. Our tool extends the most popular gas estimator, built into the Solidity compiler Remix, by accepting a broader class of contracts and by providing a list of estimates depending on conditions rather than a single upper value.

4.1 Control flow analysis for EVM bytecode

If $P = \{i_1, \ldots, i_n\}$ is a program, then a *basic block* of P is a sequence of consecutive instructions of P that are always executed consecutively. A basic block begins on the first instruction of P, or in a `JUMPDEST` instruction, or immediately after a `JUMPI` instruction; it ends in the last instruction, or in a terminating – or invalid – instruction, or in a `JUMP`/`JUMPI` instruction. The *control flow graph* (CFG) of a program P is the directed graph whose vertices are the basic blocks of P and whose edges are the pairs of blocks (u, v) such that there exists an execution of P where the execution of u is immediately followed by the execution of v. Some information about the edges of the CFG of

a program may be learned statically, but there are also some jumps whose destination can only be determined dynamically. Our tool combines both approaches, but for the theoretical analysis of the next section we will assume that we have an oracle to compute the CFG of a program available.

The gas costs of the majority of the operations are constant, so calculating their contribution to the total cost, assuming that the code has no loops, is trivial. Some operations have a small number of possible costs – for instance, `SSTORE` costs 20000 gas is a non-zero value is written in a position of the storage that was clear, and 5000 gas otherwise –, in which case the current call stack forks and the conditions of the new call stacks are updated with the assumptions made on the value to be written and on the previous content of the storage. Finally, the costs of some instructions are expressions that directly depend on their arguments – `EXP`, which receives a and b and computes $a^b \bmod 2^{256}$, has a cost of $10 + 50(1 + \log_{256} b)$ gas – and in this case the expression is added to the total gas cost.

4.2 Loop analysis

The CFG of a program, albeit very useful to reason about execution paths, does not contain important information for gas cost estimation such as the number of iterations of each circuit. Loop identification from bytecode is a topic of its own; we will explore loops with a single entry point and an algorithm to determine if a CFG is reducible. These results are based on [6], [7], [12], and [13].

We should start by defining what is known [12] as transformation T_2. T_2 is an operation over a graph $G = (V, E)$ such that, if v, w are vertices connected by an edge $(v, w) \in E$, and that is the only edge incident in w, then w is collapsed into v. The result of *collapsing* w into v in G is defined as the graph $G_0 = (V_0, E_0)$ such that $V_0 = V \setminus \{w\}$ and E_0 is E modified such that every edge of E that ends in w corresponds to an edge of E_0 with the same source and target v, and every edge of E that begins in w corresponds to an edge of E_0 with source v and the same target.

Definition 5 (Reducible graph)**.** *A graph is reducible if the successive application of the transformation T_2 results in a graph with a single vertex.*

A *depth-first spanning tree* (DFST) T of a graph G is a spanning tree of G built using depth-first search on the vertices of G. If u, v are two vertices of a graph G, and a DFST T of G is fixed, v is an *ancestor* of u (relatively to T) if there exists a path from v to u in T. In the following it is assumed that T is fixed; it can be proved that the results obtained are independent of the choice of T.

Definition 6 (Dominance)**.** *Let $G = (V, E)$ be a CFG and let $u, v \in V$. u dominates v if every path from the entry node to v contains u.*

Definition 7 (Back edge, Cycle edge)**.** *Let (V, E) be a CFG and let $u, v \in V$ be such that $(u, v) \in E$.*

- *(u, v) is a back edge if v dominates u;*
- *(u, v) is a cycle edge if v is an ancestor of u.*

It is clear that a back edge is always a cycle edge, but the converse may be false. Our primary interest will be in graphs where every cycle edge is a back edge.

Theorem 1. *$G = (V, E)$ is reducible if and only if for every $v, w \in V$ such that (v, w) is a cycle edge, w dominates v in G.*

The main idea of Tarjan's algorithm [12] is to iteratively apply collapsing operations to the vertices that are targets of cycle edges of G (intuitively, loop headers) until we either find a cycle edge that is not a back edge or we exhaust all vertices (thus proving that the graph is reducible). If the graph and the operations over the graph are implemented using a *disjoint-set data structure* it is possible to obtain fairly good results about time and space complexity. Some proofs can be found in [12] and [2], while others are only sketched and are detailed in the complete version of the thesis.

We now present two important lemmas that lead to the proof of the correction of the algorithm, where G_0 is the result of applying a contraction to G. To that end, if w is a vertex of G, we define:

$$\begin{aligned} C(w) =& \{v \in V : (v, w) \text{ is a cycle edge in G}\} \\ P(w) =& \{v \in V : \exists z \in C(w) \text{ such that there is} \\ & \qquad v \to z \text{ in } G \text{ not containing } w\} \end{aligned}$$

Lemma 1. *G is reducible if and only if for all w and for all $v \in P(w)$, there exists a path from w to v in T.*

Lemma 2. *Every edge (v_0, w_0) of the graph G_0 corresponds to an edge (v, w_0) of the graph G where v is such that there exists v_0 such that there is a path from v_0 to v in T.*

The main results on correction and complexity are stated in Propositions 1 and 2.

Proposition 1. *Tarjan's algorithm, when applied to a graph G, returns* ***true*** *if and only if G is reducible.*

Proposition 2. *Tarjan's algorithm, when given an input graph G with n vertices and m edges, has worst-case running time $O((n+m)\alpha(n))$ and requires $O(n+m)$ space.*

An important property about reducible control flow graphs is that every circuit is a *natural loop*. Natural loops correspond to typical `for` or `while` loops and contain a single entry point, which is desirable for reasoning about execution paths.

Definition 8 (Natural loop)**.** *Let $G = (V, E)$ be a CFG. A natural loop of a back edge $(u, w) \in E$ is the subgraph $G_{loop} = (V_{loop}, E_{loop})$ of G such that $V_{loop} = \{w\} \cup \{v \colon \exists v \to u$ that does not contain $w\}$ and $E_{loop} = \{(u, v) \in E \colon u, v \in V_{loop}\}$.*

Even in this context, computing the number of iterations of a loop is an NP-hard problem, so in practice we will resort to some heuristics to determine that number.

4.3 Description of the tool

The tool was designed to deal with bytecode resultant from compiling typical smart contracts written in Solidity and it runs over an adaptation of the symbolic machine explored in Section 3. In Solidity, an object-oriented language, a contract is similar to a class, and contains global variables and functions. The gas estimator receives the code and a hexadecimal string that indicates which function should be analysed, and returns a list of pairs of the form $(cost, condition)$, where *cost* may be a number or an expression depending on input, transaction or global state parameters, and *condition* is a formula.

It can handle bytecode whose control flow graph does not have circuits and bytecode whose control flow graph has *structured for loops*, which we define as natural loops with a constant number of iterations, a single exit point (the header of the loop), a guard that consists of a conditional jump depending on the comparison of an integer with a variable through one of the operators $<, \leq, >, \geq$, and the iterator variable is either an element of the storage of the contract that is currently executing or an element of the execution stack.

The CFG is built in two phases: the first one catches static jumps and the second one is an iterative process that intends to capture every possible destination of any dynamic jump. It is checked in each step that the current CFG is reducible, since in the irreducible case we are not able to provide an estimate. The CFG is finished when every leaf is a block that ends with a terminal or invalid instruction. Loop identification and analysis occurs during the iterations of the second phase; if a loop that does not comply with the restrictions that were defined previously is detected, we are also not able to give an estimate.

```
function winningProposal() public view
returns (uint wP)
{
    uint wVote = 0;
    for (uint p = 0; p < 3; p++) {
        if (proposals[p].count > wVote) {
            wVote = proposals[p].count;
            wP = p;
        }
    }
}
```

Figure 6: `winningProposal` function code

Symbolic execution of the basic blocks that constitute the body of the loop allows the tool to get to know, for each loop, the iterator variable, its increment during the execution of the body of the loop, initial and final values and, from that information, the number of iterations, which is then multiplied by the gas cost of a single iteration of the loop in order to determine its contribution to the total.

4.4 Estimation of gas costs for real smart contracts

The present section illustrates the results given by the gas estimator when used the contexts of a contract with a loop.

The contract in question is named Voting and is part of the Solidity example contracts[1]. This contract implements an electronic election where each `Voter` can vote, at most, in one `Proposal`. We will consider a modified version of the contract where the array `proposals`, a list of elements of the type `Proposal`, contains exactly 3 elements. The code can be consulted in the full document. We will estimate the gas cost of calling the function `winningProposal`, which returns the `Proposal` with the highest number of votes so far, and whose code, where some names have been shortened, can be seen in Figure 6.

Remix gives an upper estimate of "infinite" for this function since it contains a loop. Our tool returns a

[1]The code of the contract can be seen at: `https://solidity.readthedocs.io/en/v0.4.24/solidity-by-example.html`

list containing 5 pairs, which costs and conditions over `proposals[p].count`, for $0 \leq p \leq 2$ we informally describe below:

- 175 gas if the value of the transaction where the execution occurs is different from 0 – this function does not accept ether and so an exception is raised early on the execution;
- 1223 gas if no proposal has votes;
- 1482 gas if one of the proposals was once declared as the current winner and did not leave that position until the end of the execution;
- 1741 gas if one of the proposals was once declared as the current winner and was later surpassed by another proposal;
- 2000 gas if every candidate was the current winner once but the final winner was determined to be the last one.

This way, not only it is possible to know that the function will never spend more than 2000 gas, it is also made clear under which conditions the function is more costly to execute.

5 Conclusions

In the present work we have studied the Ethereum framework, focusing on the Ethereum Virtual Machine, and developed two tools designed to analyse smart contracts. The first tool allows symbolic execution of smart contracts and eases formal verification of properties related to the global state of the system. The second tool estimates the possible gas costs resultant of the execution of a broad class of smart contracts, including programs containing loops with a fixed number of iterations, which constitutes an improvement over the previous gas estimators.

Future research directions on this topic could include enhancements on the gas estimator with the goal of widening the class of programs that is accepted, and the development of an optimizer of Ethereum bytecode, since the current compilers often generate redundant and expensive code.

References

[1] V. Buterin. A next-generation smart contract and decentralized application platform. Available at: `http://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf`, 2014.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 21. MIT Press, 2nd edition, 2002.

[3] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *12th Annual International Cryptology Conference*, pages 139 – 147, 1992.

[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] C. A. Gunter. *Semantics of programming languages: structures and techniques*. MIT Press, 1992.

[6] M. S. Hecht and J. D. Ullman. Flow graph reducibility. *SIAM J. Comput.*, 1:188–202, 1972.

[7] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21:367–375, 1974.

[8] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Available at: `https://bitcoin.org/bitcoin.pdf`, 2009.

[9] H. R. Nielson and F. Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, 1992.

[10] A. Sernadas and C. Sernadas. *Foundations of logic and theory of computation*. College Publications, 2008.

[11] N. Szabo. Smart contracts: Building blocks for digital markets. Available at: `http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html`, 1994.

[12] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[13] R. E. Tarjan. Amortized computational complexity. *SIAM. J. on Algebraic and Discrete Methods*, 6:306–318, 1985.

[14] G. Wood. Ethereum: A secure decentralized generalized transaction ledger. Available at: `https://gavwood.com/paper.pdf`, 2014.