5-2020

Semantic understanding of smart contracts: Executable operational semantics of solidity

Jiao JIAO
Nanyang Technological University

Shuanglong KAN
Nanyang Technological University

Shang Wei LIN
Nanyang Technological University

David SANAN
Nanyang Technological University

Yang LIU
Nanyang Technological University

See next page for additional authors

Citation

JIAO, Jiao; KAN, Shuanglong; LIN, Shang Wei; SANAN, David; LIU, Yang; and SUN, Jun. Semantic understanding of smart contracts: Executable operational semantics of solidity. (2020). *2020 IEEE 41st Symposium on Security and Privacy: San Francisco, May 18-21: Proceedings*. 1695-1712.
Available at: https://ink.library.smu.edu.sg/sis_research/5968

Author
Jiao JIAO, Shuanglong KAN, Shang Wei LIN, David SANAN, Yang LIU, and Jun SUN

Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity

Jiao Jiao*, Shuanglong Kan*, Shang-Wei Lin*, David Sanan*, Yang Liu* and Jun Sun†
*Nanyang Technological University, Singapore and †Singapore Management University, Singapore
{jiao0023,slkan,shang-wei.lin,sanan,yangliu}@ntu.edu.sg, {junsun}@smu.edu.sg

***Abstract*—Bitcoin has been a popular research topic recently. Ethereum (ETH), a second generation of cryptocurrency, extends Bitcoin's design by offering a Turing-complete programming language called Solidity to develop smart contracts. Smart contracts allow creditable execution of contracts on EVM (Ethereum Virtual Machine) without third parties. Developing correct and secure smart contracts is challenging due to the decentralized computation nature of the blockchain. Buggy smart contracts may lead to huge financial loss. Furthermore, smart contracts are very hard, if not impossible, to patch once they are deployed. Thus, there is a recent surge of interest in analyzing and verifying smart contracts. While most of the existing works either focus on EVM bytecode or translate Solidity smart contracts into programs in intermediate languages, we argue that it is important and necessary to understand and formally define the semantics of Solidity since programmers write and reason about smart contracts at the level of source code. In this work, we develop a formal semantics for Solidity which provides a formal specification of smart contracts to define semantic-level security properties for the high-level verification. Furthermore, the proposed semantics defines correct and secure high-level execution behaviours of smart contracts to reason about compiler bugs and assist developers in writing secure smart contracts.**

I. Introduction

The success of Bitcoin since 2009 stimulates the development of other blockchain-based applications, such as Ethereum [1], a second generation of cryptocurrency which supports the revolutionary idea of smart contracts. A smart contract [2] is a computer program written in a Turing-complete programming language called Solidity, which is stored on the blockchain to achieve certain functionality. Smart contracts benefit from the features of the blockchain in various aspects. For instance, it is not necessary to have an external trusted authority to achieve consensus, and transactions through smart contracts are always traceable and credible.

Smart contracts must be verified for multiple reasons. Firstly, due to the decentralized nature of the blockchain, smart contracts are different from programs written in other programming languages (e.g., C/Java). For instance, the storage of each contract instance is at a permanent address on the blockchain. In this way, each instance is a particular execution context and context switches are possible through external calls. Particularly, `delegatecall` is executed in the context of the caller rather than the recipient, making it possible to modify the caller state. Programming smart contracts thus is error-prone without a proper understanding of the underlying semantic model. This is further worsened by multiple language design choices (e.g., fallback functions) made by Solidity. To understand the execution behaviors of smart contracts, we must understand the semantics of Solidity, and make sure that it is formally defined so that programmers can write contracts accordingly. If a programmer implements a smart contract with his/her understanding inconsistent with the Solidity semantics, vulnerabilities are very likely to be introduced. Secondly, verifying smart contracts against vulnerabilities in deployed contracts is crucial for protecting digital assets. One well-known attack on smart contracts is the DAO attack [3]. The attacker exploited a vulnerability associated with fallback functions and the reentrancy property [4] in the DAO contract, and managed to take 60 million dollars under his control. Finally, unlike traditional software which can be patched, it is very hard if not impossible to patch a smart contract once it is deployed due to the very nature of the blockchain. For instance, the team behind Ethereum intended to conduct a hard fork of the Ethereum network in view of the DAO attack, which turned out to be controversial [5]. It is thus extremely important that a smart contract has been verified before it is deployed on the blockchain.

A. Related Works

There is a surge of interest in analyzing and verifying smart contracts. Some of the existing works conduct verification or security analysis on EVM bytecode [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26]. For instance, Oyente [6] is a symbolic execution engine which targets bytecode running on Ethereum Virtual Machine (EVM). In addition, KEVM [7] is a semantic encoding of EVM bytecode in the K-framework to facilitate the formal verification of smart contracts at bytecode level. A set of test oracles is defined in [8] to detect security vulnerabilities at EVM level. In [11], a semantic framework is proposed to analyze smart contracts based on a small-step EVM semantics. Securify [12] translates EVM bytecode into a stackless representation in static-single assignment form to infer semantic facts for analyzing smart contracts. In other works, the verification or security analysis of smart contracts is based on intermediate languages [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37]. For instance, the formalization in F* [27] is an intermediate-level language for the equivalence checking of Solidity programs and EVM bytecode. In addition, Zeus [28] translates Solidity contracts into programs in an abstract language which are translated into LLVM bitcode to verify smart contracts. Similarly, Solidity programs are translated into Boogie programs to be analyzed in the proposed verifiers in [29], [35]. The online detection of Effectively Callback Free (ECF) [30] objects is facilitated by a simple imperative object-based programming language, called SMAC.

To the best of our knowledge, most of the existing works either focus on EVM bytecode, or translate Solidity contracts into programs in intermediate languages which are suitable for verifying smart contracts or detecting potential issues in associated verifiers or checkers. We believe that it is necessary and also important to formally define the semantics of Solidity. The first reason is that programmers write and reason about smart contracts at the level of source code without the semantics of which they are required to understand how Solidity programs are compiled into bytecode to understand contracts, which is far from trivial. Secondly, even though Solidity contracts can be transformed into programs in intermediate languages to be analyzed and verified in existing model checkers and verifiers, the equivalence checking of Solidity and the intermediate language considered is crucial to the validity of the verification. For instance, most of the false positives reported in Zeus [28] are introduced by the semantic inconsistency of Solidity and the abstract language. Finally, although alternative high-level smart contract languages, such as Vyper [38], Bamboo [39], Flint [40], etc, have been proposed for safer programming, Solidity is still the most popular language for writing smart contracts. According to the data obtained from Etherscan [41], with respect to 500 contracts deployed from 30 Nov 2019 to 8 Dec 2019, all of them are written in Solidity.

Lolisa [42] is a formal semantics of a subset of Solidity designed based on a formal memory (GERM) framework. This semantics is not constructed directly on Solidity, which introduces equivalence checking issues (i.e., the correctness and completeness of the semantic transformation from the Solidity features to the ones supported by the GERM model) for further validation. Furthermore, a big-step semantics of a small subset of Solidity is formalized in [43]. However, this formalization fails to address many important Solidity features, such as function overloading, integers of sizes less than 256 bits, packed byte array types, libraries, events, etc. Featherweight Solidity [44] is a calculus formalizing the core features of Solidity with a static type system. A refinement of the type system is also proposed to enhance the type safety of Solidity. Similarly, TinySol [45] is a minimal calculus for Solidity contracts. Nevertheless, these calculus-based formalizations are not directly executable, making them infeasible in semantics validation and automatic verification. Therefore, a complete direct executable formal semantics of Solidity is a must in both understanding and verifying smart contracts.

B. Challenges

The challenges of developing Solidity semantics lie in two aspects. Firstly, there are insufficient documentations defining or describing the complete features of Solidity. For instance, the official documentation [46] introduces each feature with a few examples, from the perspective of which, it is difficult for readers to fully understand the complete features of the language since only a part of the semantics is involved in the examples illustrated. Furthermore, these documentations fail to address corner cases including ambiguities and undefined behaviours. This requires that the Solidity semantics has to be executable for extensive testing to eliminate ambiguities and resolve undefined behaviours. Another way to understand the semantics is to figure out how Solidity compilers, such as Remix [47], interpret it by observing the execution behaviours. However, this may be problematic since these compilers only capture low-level instructions which may be inconsistent with the high-level semantics. Therefore, in addition to taking the execution behaviors observed from the compilers as "benchmarks" to interpret the semantics, it is necessary to define the correct semantics from the perspective of what it should be to get rid of the misunderstandings introduced by the compilers.

The second aspect of the challenges comes from the language features. The decentralized nature of the blockchain makes the Solidity features unique. As mentioned above, smart contracts work in a distributed mode, and each contract instance has a storage located at a permanent address on the blockchain, which makes inter-contract calls possible. Therefore, the values of state variables in a contract instance can be modified by another one through external calls to this instance. Many potential issues can be triggered by the untrusted nature of calls. For instance, in the DAO attack [3], the late update of state variables makes it possible to enter the associated function for a second time, resulting in ether loss. However, this would never be possible for programs written in other languages, such as C or Java, without such distributed nature. Furthermore, the blockchain related features in Solidity, such as `mappings` associated with accounts, `fallback functions`, transaction-based `assert` and `revert`, etc, add to the difficulty of the semantics design since there are no standard interpretations that can be borrowed from other language semantics to take into account the blockchain context. Apart from uniqueness, Solidity supports a variety of calls, such as high-level and low-level calls, constructors, fallback functions, etc, possibly with ether transfer, in different formats of syntax, and exception handling features, such as `assert`, `revert`, `require`, etc. A uniform mechanism is necessary to construct the semantics to make it extensible. Lastly, the evolution of the Solidity language features, such as the reformatting of `constructor` and the deprecation of `throw`, makes the semantics design even more difficult. In order to survive in the language evolution, Solidity semantics must be designed from a very general point of view, which also makes it extensible for alternative smart contract languages.

C. Contributions

In this work, we develop an executable operational semantics for the Solidity programming language to formally reason about smart contracts written in Solidity. The contributions of this work lie in four aspects. Firstly, our work is the first approach, to our knowledge, to a complete executable formal semantics of Solidity constructed directly on the language itself other than Solidity compilers. The proposed executable semantics completely covers the supported high-level core features specified by the official Solidity documentation [46] and is validated with the official compiler Remix [47]. In addition, a new and general way of semantics formalization is applied in the semantics design, making the proposed semantics robust in the language evolution of smart contracts. Secondly, the proposed semantics provides a formal specification of smart contracts which solves the specification issues in the existing verification and analysis tools. Thirdly, the proposed semantics allows us to formally define semantic-level security properties for verifying smart contracts to exclude the false positives introduced by the existing approaches. Finally, the proposed semantics defines correct and secure high-level execution behaviours of smart contracts to reason about compiler bugs and assist developers in writing secure smart contracts.

D. Outline

The remaining part of this paper is organized as follows. Section II introduces the background of smart contracts and the K-framework. The proposed executable operational semantics of Solidity formalized in the K-framework is introduced in Section III. Section IV shows the evaluation results of the proposed semantics. The applications of the Solidity semantics are introduced in Section V. Section VI concludes this work.

II. Preliminaries

In this section, we briefly introduce the background of smart contracts and the K-framework.

A. Smart Contracts

Ethereum [1], proposed in late 2013 by Vitalik Buterin, is a blockchain-based distributed computing platform supporting smart contract functionality. It provides a decentralized international network where each participant node equipped with EVM can execute smart contracts. It also provides a cryptocurrency called “ether” (ETH), which can be transferred between different accounts and used to compensate participant nodes for their computations on smart contracts.

Solidity is one of the high-level programming languages to implement smart contracts on Ethereum. A smart contract written in Solidity can be compiled into EVM bytecode and then be executed by any participant node equipped with EVM. A Solidity smart contract is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain [46]. Fig. 1 shows an example of Solidity smart contracts, named `Coin`, implementing a very simple cryptocurrency. In line 2, the public state variable `minter` of type `address` is declared to store the address of the minter of the cryptocurrency, i.e., the owner of the smart contract. The constructor, denoted by `constructor()`, is defined in lines 5–7. Once the smart contract is created and deployed[1], its constructor is invoked automatically, and `minter` is set to be the address of its creator (owner), represented by the built-in keyword `msg.sender`. In line 3, the public state variable `balances` is declared to store the balances of users. It is of type `mapping`, considered as a hash-table mapping from keys to values. In this example, `balances` maps from a user (represented as an address) to his/her balance (represented as an unsigned integer value). The `mint` function, defined in lines 9–13, is supposed to be invoked only by its owner to mint coins, the number of which is specified by `amount`, for the user located at the `receiver` address. If `mint` is called by anyone except the owner of the contract, nothing will happen because of the guarding `if` statement in line 11. The `send` function, defined in lines 15–20, can be invoked by any user to transfer coins, the number of which is specified by `amount`, to another user located at the `receiver` address. If the balance is not sufficient, nothing will happen because of the guarding `if` statement in line 17; otherwise, the balances of both sides will be updated accordingly.

```
1  contract Coin {
2     address public minter;
3     mapping (address => uint) public balances;
4  
5     constructor() public {
6        minter = msg.sender;
7     }
8  
9     function mint(address receiver, uint amount) public
10    {
11       if (msg.sender != minter) return;
12       balances[receiver] += amount;
13    }
14 
15    function send(address receiver, uint amount) public
16    {
17       if (balances[msg.sender] < amount) return;
18       balances[msg.sender] -= amount;
19       balances[receiver] += amount;
20    }
21 }
```

Figure 1. Solidity Smart Contract Example

A blockchain is actually a globally-shared transactional database or ledger. Every participant node can read the information on the blockchain. If one wants to make any state change on the blockchain, he or she has to create a so-called *transaction* which has to be accepted and validated by all other participant nodes. Furthermore, once a transaction is applied to the blockchain, no other transactions can alter it. For example, deploying the `Coin` smart contract generates a transaction because the state of the blockchain is going to be changed, i.e., one more smart contract instance will be included. Similarly, any invocation of the function `mint` or `send` also generates a transaction because the state of the contract instance, a part of the whole blockchain, is going to be changed. Transactions have to be selected and added into blocks to be appended to the blockchain. Appending blocks to Proof-of-Work (PoW) [48] blockchains involves solving a computationally challenging mathematical problem. This procedure is the so-called *mining*, and the participant nodes are called *miners*.

B. The K-framework

The K-framework ($\mathbb{K}$) [49] is a rewriting logic [50] based formal *executable* semantics definition framework. The semantics definitions of various programming languages have been

[1] How to create and deploy a smart contract is out of scope and can be found in: https://solidity.readthedocs.io

$$\left\langle \left\langle \texttt{\$PGM:Pgm} \right\rangle_k \left\langle \texttt{.Map} \right\rangle_{state} \right\rangle_T$$

Figure 2. IMP Configuration

```
Pgm ::= "int" Ids ";" Stmt  Ids ::= List{Id, ","}
AExp ::= Int | Id | "-" Int | AExp "/" AExp
       > AExp "+" AExp | "(" AExp ")"
BExp ::= Bool | AExp "<=" AExp | "!" BExp
       > BExp "&&" BExp | "(" BExp ")"
Block ::= "{" "}" | "{" Stmt "}"
Stmt ::= Block | Id "=" AExp ";"
       | "if" "(" BExp ")" Block "else" Block
       | "while" "(" BExp ")" Block > Stmt Stmt
```

Figure 3. The Syntax of IMP

developed using $\mathbb{K}$, such as Java [51], C [52], etc. Particularly, an executable semantics of EVM (Ethereum Virtual Machine) [7], the bytecode language of smart contracts, has been constructed in the K-framework. $\mathbb{K}$ backends, like the Isabelle theory generator, the model checker, and the deductive verifier, can be utilized to prove properties on the semantics and construct verification tools [53].

A language semantics definition in the K-framework consists of three main parts, namely the language syntax, the configuration specified by the developer and a set of rules constructed based on the syntax and the configuration. Given a semantics definition and some source programs, the K-framework executes the source programs based on the semantics definition. In addition, specified properties can be verified by the formal analysis tools in $\mathbb{K}$ backends. We take IMP [54], a simple imperative language, as an example to show how to define language semantics in the K-framework.

The configuration of the IMP language is shown in Fig. 2. There are only two cells, namely `k` and `state` in the whole configuration cell `T`. The cells in the configuration are used to store some information related to the program execution. For instance, the cell `k` stores the source program for execution `Pgm`. Furthermore, the cell `state` records the mapping from a variable name to its value.

Here, we introduce some basic rules in the K-IMP semantics. These rules are `allocate`, `read` and `write` for variables. The syntax of IMP is also given in Fig. 3.

Let us start with the rule of memory allocations for variables in IMP shown in ALLOCATE. When `Pgm`, interpreted as `int X,Xs;S`, is encountered, we need to store a list of variables (`X,Xs`) starting from `X` in the cell `state` with a list of mappings. Here `state` can be regarded as a physical memory or storage, and `Xs` is also a list of variables which can be empty. X is popped out of the cell `k` and a new mapping from X to 0 is created in the cell `state`, which means that a memory slot has been allocated for X to store its initial value 0. No duplicate names are allowed in `state`, which is guaranteed by the require condition. Then we go like this until `Xs` becomes empty, which means that all the variables have already been stored in `state`. At this point, the execution of the first part of `Pgm` has been finished and we proceed to the execution of the statement `S`. This can be summarized in FINISH-ALLOCATE where `.Ids` is an empty list of identifiers, which means that the variable list is empty. Please note that `.` means an empty set in the K-framework. If a rule ends with `.`, it means that nothing will be executed.

RULE ALLOCATE

$$\left\langle \frac{\texttt{int X,Xs;S}}{\texttt{int Xs;S}} \cdots \right\rangle_k \left\langle \frac{\texttt{Rho:Map}}{\texttt{Rho (X |-> 0)}} \right\rangle_{state}$$

`requires notBool (X in keys(Rho))`

RULE FINISH-ALLOCATE

$$\left\langle \frac{\texttt{int .Ids;S}}{\texttt{S}} \cdots \right\rangle_k$$

RULE READ

$$\left\langle \frac{\texttt{X:Id}}{\texttt{I}} \cdots \right\rangle_k \left\langle \cdots \ \texttt{X |-> I} \ \cdots \right\rangle_{state}$$

RULE WRITE

$$\left\langle \frac{\texttt{X = I:Int;}}{\texttt{.}} \cdots \right\rangle_k \left\langle \cdots \frac{\texttt{X |-> \_}}{\texttt{X |-> I}} \cdots \right\rangle_{state}$$

Then we come to the rules of `read` and `write`. As shown in READ, if we want to look up the value of the variable `X`, we need to search it in the cell `state` by mapping the variable name `X` to its value `I`. So the evaluation of this expression `X` is its value `I`. If we cannot find a mapping for `X`, the program execution will stop at this point. Particularly, `...` means there can be something in the corresponding position. For instance, the mapping of `X` can be in any position in the cell `state`. However, in the cell `k`, `...` can only be at the end since the program stored in `k` is executed sequentially. As illustrated in WRITE, if we want to assign the integer value `I` to the variable `X`, similarly we need to search it in the cell `state` by mapping the variable name. However, we also need to rewrite the value of X, denoted by "_" which is a placeholder, to `I`.

As illustrated above, formalizing language semantics in the K-framework is really specific to its configuration which stores the information related to the program execution, such as variables, statements, etc. Firstly, cells in the configuration can be regarded as physical features, e.g., memory or storage, or logical concepts, such as stacks, mappings, etc, which may be more common in complex languages. Secondly, cell contents can be either mapped or rewritten. When we map contents in a cell, this cell is used to formalize a condition requirement. Only when the contents in the cell match the desired ones, the rule involving this cell with the specific contents can be applied (e.g., mapping the name of the variable `x` in READ). Also, cell contents can be rewritten to stay updated with the program execution (e.g., rewriting the value of the variable `x` in WRITE). Finally, cell contents can be retrieved to facilitate the program execution. For instance, function definitions are stored in a list of cells. When executing function calls, we need to retrieve the function definitions that have been stored to get the information about the functions to be called.

III. Formal Semantics of Solidity in the K-framework

In this section, we introduce the executable operational semantics of Solidity formalized in the K-framework. The syntax of this semantics is constructed based on the official Solidity documentation [46]. The configuration is designed specifically for Solidity smart contracts. Based on the syntax and the configuration, we formalize the operational rules for the language features with rewriting logic.

A. Runtime Configuration of Solidity

Due to limit of space, the configuration in Fig. 4 is shown from a general point of view and some of the sub-cells are omitted. In this configuration, there are six main cells in the whole configuration cell `T` and they are `k`, `controlStacks`, `contracts`, `functions`, `contractInstances` and `transactions`. The value of each cell is initialized in the configuration with its type specified. A dot followed by any type represents an empty set of this type. For instance, `.List` is an empty list. Particularly, `K` is the most general type which can be any specific type defined in the K-framework.

In the cell `k`, the source programs, called SourceUnit, are stored for execution. If the programs stored in `k` terminate in a proper way, there will be a dot in this cell, indicating that this cell is empty and there are no more programs to execute.

The cell `controlStacks` records `contractStack`, `functionStack`, `newStack` and `blockStack`. To be specific, `contractStack`, `functionStack`, `newStack` and `blockStack` keep track of lists of contract instances, function calls, new contract instance creations and variable contexts to look up and assign values to variables in different scopes with the current ones on the top, respectively.

In the cell `contracts`, a set of contract definitions is stored. Each cell `contract` represents a contract definition. The number of distinct contracts is counted in `cntContractDefs`. In `contract`, the contract name is stored in `cName`. State variable information is stored in `stateVars`. In addition, `Constructor` indicates whether the contract has a constructor or not and its initial value is false. The constructor of a Solidity smart contract is an optional function declared with the keyword "constructor" which is executed for once when a new instance is created [46].

Similarly, the cell `functions` stores a set of function definitions. Each cell `function` represents a function definition. The total number of function definitions is stored in `cntFunctions`. For each function definition, the function Id and name are stored in `fId` and `fName`, respectively. In addtion, function parameters, including input parameters and return parameters, are recorded in the corresponding cells. We also store the function body in `Body` and function quantifiers which can be modifiers or specifiers in `funQuantifiers`.

In the cell `contractInstances`, there is a set of contract instances. Each cell `contractInstance` represents a contract instance. The number of contract instances is counted in `cntContracts`. We store the contract instance Id and the name of its associated contract in `ctId` and `ctName`, respectively. Four different mappings are applied to store more information of a variable. Specifically speaking, `ctContext`, `ctType`, `ctLocation` and `ctStorage/Memory` record the mappings from a variable name to its logical address in the storage or memory, a variable name to its type, a variable name to its location information, namely "global" or "local", and the logical address of a variable in the storage or memory to its value, respectively. `globalContext` records the state variable context. The number of memory slots taken by variables is calculated in `slotNum`. `Balance` records the balance of each contract instance.

In `transactions`, we record the number of transactions in `cntTrans`, every transaction in `tranComputation` and also "msg" information in `Msg` and `msgStack`. "msg" is a keyword in Solidity to represent transaction information. For instance, "msg.sender" is the caller of the function and "msg.value" specifies the amount of ether to be transferred. `msgStack` stores a list of transaction information tuples at increasing call depths while `Msg` records the current one. We simulate transactions of smart contracts with a "Main" contract similar to the main function in C where new contract instances can be created and external function calls to these instances are available. The Id of the "Main" contract is "-1", since other contract instances start from 0. Therefore, the initialized content in `contractStack` is `ListItem(-1)`, and `cntTrans` is counted from 1, indicating that the creation of the "Main" contract is the first transaction recorded in `tranComputation.` `gasConsumption` records the amount of gas consumed in the current call, and `gasStack` stores gas consumption at increasing call depths.

B. Semantics of the Core Features

We construct Solidity semantics from a very general point of view to adapt to the language evolution of smart contracts. To be specific, instead of directly using the formats of function calls in Solidity, all kinds of function calls, including high-level and low-level calls, constructors and fallback functions, are rewritten to a uniform format. In this way, the semantics is extensible by inheriting commonly shared parts and adding specific parts, making it possible to interpret the semantics of other high-level smart contract programming languages with the proposed Solidity semantics as long as their core semantic features fall into the ones of Solidity. In the following part of this section, we present an overview of three core semantic features in Solidity, namely memory operations, new contract instance creations and function calls. Particularly, new contract instance creations and function calls are the two kinds of transactions on the blockchain. Due to limit of space, common features, such as loops, arithmetic operations, etc, and implementation details of sub-steps are omitted.

Types are specified in the presented rules. `Id` stands for identifiers and `Int` represents integers. `EleType` represents elementary types in Solidity, such as `int`, `uint`, `address`, etc. `Values` is a list of `Value` types which can be integers (`Int`) or Boolean types (`Bool`). `Msg` is the type of transaction information. `Map` represents mappings. `ExpressionList` is a list of expressions.

1) Memory Operations: We present the semantics rules for memory operations on elementary types in Solidity, such as `int`, `uint` and `address`, each of which takes only one

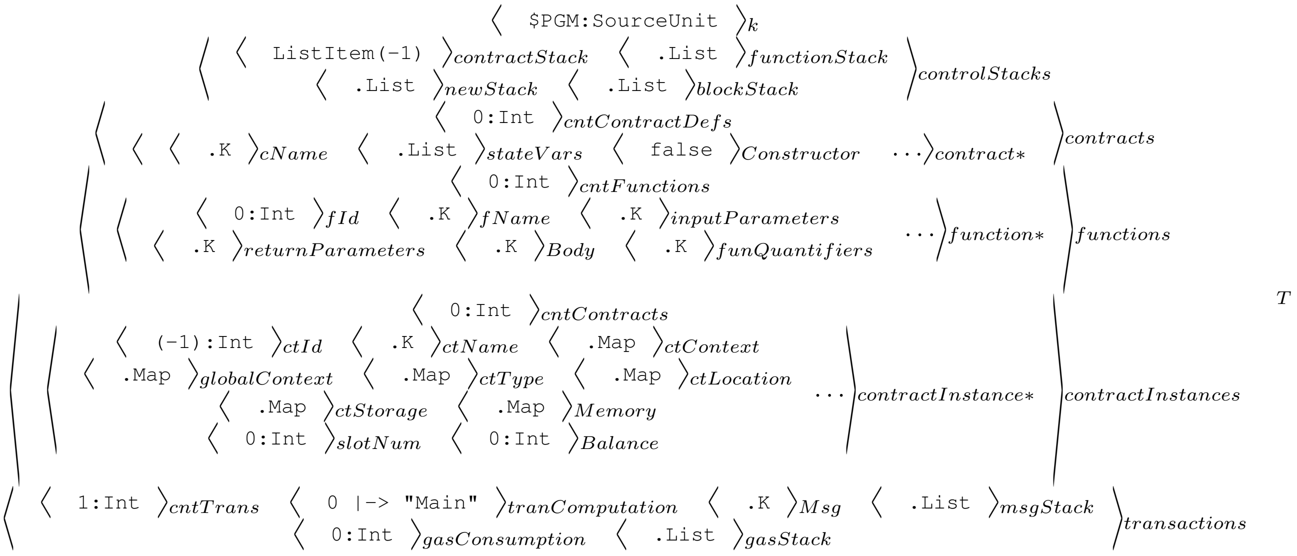

Figure 4. Runtime Configuration of Solidity

memory slot. Complex types, such as arrays, mappings, etc, are compositions of elementary types. A memory operation on a complex type can be regarded as a set of recursive memory operations on elementary types. For instance, the memory allocation for a one-dimensional fixed-size array is equivalent to allocating an elementary type for each index of this array. Reading and writing a particular index involve recursive steps to retrieve the logical address of this index from the base address of the array. Mappings are similar to dynamic arrays.

Let us start with the `read` operation on elementary types shown in READ. Here, we consider the object as a variable, denoted by `X`, which is an `Id` type. The first thing to do is to get the current execution context. This is achieved by retrieving the current contract instance Id `N` in `contractStack` and mapping the corresponding contract instance with `N` in the cell `ctId`. After that, we retrieve the logical address of `X`, denoted by `Addr`, in `ctContext` and the location information of `X`, denoted by `L`, in `ctLocation`. With these two parameters, we can obtain the evaluation of `X` through `readAddress` which retrieves the value located at `Addr` in the associated cell specified by `L` (cf. Appendix A for details). To be specific, if `L` specifies this variable as a global one, the search space is `ctStorage`. Otherwise, the value is retrieved in `Memory`.

`write` is similar to `read`. After retrieving the logical address of `X`, denoted by `Addr`, and the location information of `X`, denoted by `L`, we rewrite the value located at `Addr` to the value `V` in the cell specified by `L` through `writeAddress` (cf. Appendix B for details).

Then we come to the allocation for elementary types shown in ALLOCATE. The first input parameter `N` indicates the contract instance Id. The information of the variable including its name `X`, type `T`, location information `L` and initial value `V`, is stored in `#varInfo`. First, we map the corresponding contract instance with the instance Id `N` which is indicated in `ctId`. Then the number of memory slots is increased by 1 in `slotNum`. After that, the variable information is recorded in the associated cells. To be specific, we record the logical address `Addr`, the type `T`, and the location information `L` in `ctContext`, `ctType` and `ctLocation`, respectively. Finally, a memory slot is allocated for this variable through `allocateAddress` (cf. Appendix C for details).

2) New Contract Instance Creations: The semantics rule for creating a new contract instance is shown in NEW-CONTRACT-INSTANCE-CREATION. Creating a new contract instance in Solidity is achieved through `new X:Id (E:ExpressionList)` where `X` is the contract name and `E` specifies the arguments in the constructor. There are altogether three sub-steps for this transaction and they are `updateState`, `allocateStorage` and `initInstance`. The symbol $\curvearrowright$ here means “followed by”. To be specific, `updateState` updates the blockchain states, including the states of contract instances and transactions, and the stack information to indicate that we are in the process of a new contract instance creation (cf. Appendix D for details). In addition, `allocateStorage` allocates state variables (cf. Appendix E for details) and `initInstance` deals with initialization issues, such as calling the constructor, in the new contract instance (cf. Appendix F for details).

3) Function Calls: Function calls in Solidity are written in a format similar to member access. For instance, `target.deposit.value(2)()` is a typical function call in Solidity. To be specific, `target` specifies the recipient instance and `deposit` is the function to be called in that instance. `value` specifies `msg.value` as 2. In addition, we can specify other parameters, such as `msg.gas`, function arguments, etc, in the function call. In order to make the semantics of function calls general for all kinds of calls and extensible for different kinds of smart contract languages, a uniform format of function calls is applied to generalize the semantics. The uniform format is `functionCall(Id_of_Caller;`

RULE Read

$$\left\langle \frac{\texttt{X:Id}}{\texttt{readAddress(Addr,L)}} \cdots \right\rangle_{k}$$
$$\left\langle \texttt{ListItem(N:Int)}\ \cdots \right\rangle_{contractStack}$$
$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId} \\ \langle\cdots\ \texttt{X |-> Addr}\ \cdots\rangle_{ctContext} \\ \langle\cdots\ \texttt{X |-> L}\ \cdots\rangle_{ctLocation} \\ \langle\cdots\ \texttt{X |-> T:EleType}\ \cdots\rangle_{ctType} \end{array} \cdots \right\rangle_{contractInstance}$$

RULE Write

$$\left\langle \frac{\texttt{X:Id = V:Value}}{\texttt{writeAddress(Addr,L,V)}} \cdots \right\rangle_{k}$$
$$\left\langle \texttt{ListItem(N:Int)}\ \cdots \right\rangle_{contractStack}$$
$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId} \\ \langle\cdots\ \texttt{X |-> Addr}\ \cdots\rangle_{ctContext} \\ \langle\cdots\ \texttt{X |-> L}\ \cdots\rangle_{ctLocation} \\ \langle\cdots\ \texttt{X |-> T:EleType}\ \cdots\rangle_{ctType} \end{array} \cdots \right\rangle_{contractInstance}$$

RULE Allocate

$$\left\langle \frac{\begin{array}{c}\texttt{allocate(N:Int, \#varInfo(X:Id,} \\ \texttt{T:EleType, L:Id, V:Value))}\end{array}}{\texttt{allocateAddress(N,Addr,L,V)}} \cdots \right\rangle_{k}$$
$$\left\langle \begin{array}{c} \langle\ \texttt{N}\ \rangle_{ctId} \\ \left\langle \frac{\texttt{Addr}}{\texttt{Addr +Int 1}} \right\rangle_{slotNum} \\ \left\langle \frac{\texttt{CONTEXT:Map}}{\texttt{CONTEXT (X |-> Addr)}} \right\rangle_{ctContext} \\ \left\langle \frac{\texttt{TYPE:Map}}{\texttt{TYPE (X |-> T)}} \right\rangle_{ctType} \\ \left\langle \frac{\texttt{LOCATION:Map}}{\texttt{LOCATION (X |-> L)}} \right\rangle_{ctLocation} \end{array} \cdots \right\rangle_{contractInstance}$$

RULE New-Contract-Instance-Creation

$$\left\langle \frac{\texttt{new X:Id (E:ExpressionList)}}{\begin{array}{c}\texttt{updateState(X)} \curvearrowright \texttt{allocateStorage(X)} \curvearrowright \\ \texttt{initInstance(X,E)}\end{array}} \cdots \right\rangle_{k}$$

RULE Decompose-Solidity-Call

$$\left\langle \frac{\begin{array}{c}\texttt{\#memberAccess(R:Int,F:Id)} \curvearrowright \texttt{Es:Values} \curvearrowright \\ \texttt{MsgValue:Int} \curvearrowright \texttt{MsgGas:Int}\end{array}}{\begin{array}{c}\texttt{functionCall(C;R;F;Es;} \\ \texttt{\#msgInfo(C,R,MsgValue,MsgGas))}\end{array}} \cdots \right\rangle_{k}$$
$$\left\langle \texttt{ListItem(C:Int)}\ \cdots \right\rangle_{contractStack}$$

RULE Function-Call

$$\left\langle \frac{\begin{array}{c}\texttt{functionCall(C:Int;R:Int;} \\ \texttt{F:Id;Es:Values;M:Msg)}\end{array}}{\begin{array}{c}\texttt{switchContext(C,R,F,M)} \curvearrowright \\ \texttt{functionCall(F;Es)} \curvearrowright \texttt{returnContext(R)}\end{array}} \cdots \right\rangle_{k}$$

`Id_of_Recipient; Function_Name; Arguments; Msg_Info)`. Particularly, `Msg_Info` represents the transaction information stored in `#msgInfo`, including the Ids of the caller and the recipient instances, `msg.value` and `msg.gas`. The semantics rule for function calls based on this format is shown in Function-Call. When it comes to the semantics of function calls in Solidity, the first thing to do is to decompose the member access like format and transform the call into the one in the uniform format. As shown in Decompose-Solidity-Call, each decomposed part in Solidity calls is reorganized in `functionCall`. Specifically speaking, `#memberAccess(R:Int,F:Id)` specifies the recipient instance `R` and the function to be called in this instance `F`. `Es` specifies the function arguments. `MsgValue` and `MsgGas` represent `msg.value` and `msg.gas`, respectively.

The semantics of function calls is designed from a general point of view. Each external function call is regarded as an extension of an internal function call. Whenever there is an external function call, we first switch to the recipient instance and then call the function in this instance as an internal call. Finally, we switch back to the caller instance. In this way, external function calls can be achieved through internal function calls and switches of contract instances. This mechanism also applies to internal function calls where the caller instance is the same as the recipient instance.

There are three sub-steps in Function-Call. The first one is to switch to the recipient instance from the caller through `switchContext` (cf. Appendix G for details). The second is an internal function call `functionCall` (cf. Appendix H for details). The last one is to return to the caller instance through `returnContext` (cf. Appendix I for details).

Particularly, the semantics of function calls is equipped with exception handling features. Generally speaking, if an exception is encountered in an inner call, it will be propagated to the transactional function call to revert the whole transaction. The propagation of exceptions is a sub-step in `returnContext` (cf. Appendix I for details). The exception handling mechanism is also general, making it possible to deal with all kinds of exception handling features in Solidity, such as `revert`, `assert`, etc, in a similar way. This is the only optimization in the proposed semantics with respect to the official Solidity documentation [46]. It generally follows but slightly differs from the semantics of exception handling defined in the documentation since it allows the propagation of exceptions in low-level calls intended by developers. The semantics rules for exception handling are shown in Exception-Propagation and Transaction-Reversion.

There are two stages in handling exceptions. The first one is the propagation of exceptions to the function call whose caller is the "Main" contract as shown in Exception-

RULE Exception-Propagation

$$\left\langle \frac{\texttt{exception()}}{\texttt{updateExceptionState()}} \cdots \right\rangle_{k}$$
$$\left\langle \texttt{ListItem(R)ListItem(C)}\ \cdots \right\rangle_{contractStack}$$
$$\texttt{requires C >=Int 0}$$

RULE Transaction-Reversion

$$\left\langle \frac{\texttt{exception()}}{\texttt{updateExceptionState()} \curvearrowright \texttt{revertState()}} \cdots \right\rangle_{k}$$
$$\left\langle \texttt{ListItem(R)ListItem(-1)} \right\rangle_{contractStack}$$

RULE Revert

$$\left\langle \frac{\texttt{revert(.ExpressionList);}}{\texttt{exception()}} \cdots \right\rangle_{k}$$

RULE Assert

$$\left\langle \frac{\texttt{assert(true);}}{\texttt{.}} \cdots \right\rangle_{k} \quad \left\langle \frac{\texttt{assert(false);}}{\texttt{exception()}} \cdots \right\rangle_{k}$$

RULE Require

$$\left\langle \frac{\texttt{require(true);}}{\texttt{.}} \cdots \right\rangle_{k} \quad \left\langle \frac{\texttt{require(false);}}{\texttt{exception()}} \cdots \right\rangle_{k}$$

RULE Out-of-Gas

$$\left\langle \frac{\texttt{S:Statement}}{\texttt{exception()}} \cdots \right\rangle_{k} \quad \left\langle\ \texttt{\#msgInfo(\_,\_,\_,GasLimit)}\ \right\rangle_{Msg}$$
$$\left\langle\ \texttt{GasC}\ \right\rangle_{gasConsumption}$$
$$\texttt{requires GasC >Int GasLimit}$$

PROPAGATION, and the second is the reversion of the transaction as shown in TRANSACTION-REVERSION. Please note that in the first stage exceptions are propagated from inner calls to the transactional function call, while the second stage is only present in the transactional function call stemming from the "Main" contract. In the stage of propagating exceptions, the exception state is updated through `updateExceptionState()` (cf. Appendix J for details) to indicate that an exception has been encountered. In particular, the Id of the caller instance should be larger than or equal to 0 since the caller cannot be the "Main" contract in this case. And in the stage of reverting transactions, the caller is the "Main" contract whose Id is "-1". In addition to updating the exception state, the whole transaction is reverted through `revertState()` (cf. Appendix J for details). Exception handling features, such as `revert`, `assert`, `require`, etc, and out-of-gas exceptions can be interpreted with the semantics of `exception()`. The semantics rules for `revert`, `assert`, `require` and out-of-gas exceptions are shown in REVERT, ASSERT, REQUIRE and OUT-OF-GAS, respectively.

The semantics rules for function calls apply to all kinds of calls in Solidity, including high-level and low-level calls, constructors and fallback functions. For instance, if there is no function name specified or the specified name does not match any existing function in the recipient instance, the first decomposed part in DECOMPOSE-SOLIDITY-CALL will be `#memberAccess(R:Int,String2Id("fallback"))` where `R` is the Id of the recipient instance and "fallback" refers to the fallback function in that instance. In this case, the fallback function in `R` will be invoked. In addition, in the case of `delegatecall`, the recipient instance `R` is the same as the caller instance `C` since the execution takes place in the caller's context.

IV. SEMANTICS EVALUATION

The proposed executable Solidity semantics is available at https://github.com/kframework/solidity-semantics. We evaluate the proposed Solidity semantics from two perspectives: the first one is its coverage (i.e., completeness), and the second is its correctness (i.e., consistency with Solidity compilers). In this section, we show that the proposed semantics completely covers the supported high-level core language features specified by the official Solidity documentation [46] and is consistent with the official Solidity compiler Remix [47].

We evaluate and test the proposed Solidity semantics with the Solidity compiler test set [55] which consists of different test programs for each feature. The Solidity compiler test set is regarded as a standard test set or benchmarks for evaluating Solidity semantics since the test programs are written in a standard or correct way defined by the language developers and cover all the features in Solidity. There are altogether 482 tests in the Solidity compiler test set. We skip the 18 tests for `inline assembly statements` and list the number of tests for some important features in the remaining 464 tests used in the evaluation in Fig. 5. Specifically speaking, four kinds of features, namely `types`, `functions`, `statements` and `new expression`, are listed in Fig. 5. For types, we list the number of tests for `elementary types`, `struct`, `mapping`, and `array`. For statements, we list the number of tests for `if statements`, `loops`, `return`, `throw`, `assert`, `revert` and `require`. As indicated in Fig. 5, `Uint` is the most common type and `return` is the most common statement in all the tests considered. In addition, function related features, including function definitions and function calls, are present in almost all the tests considered (with a ratio of 459/464). The evaluation is done by manually comparing the execution behaviours of the semantics definition in the K-framework with the ones of the Remix compiler. We consider the proposed semantics is correct if the execution behaviours in the K-framework are consistent with the ones of the Remix compiler. A feature is considered to be fully covered if all the Solidity compiler tests involving this feature are passed. In order to adapt to the optimization of exception handling in the proposed semantics, we manually propagate exceptions in low-level calls by adding assertions in the test programs. We list the coverage of the proposed Solidity semantics in Table I from the perspective of each feature specified by the official Solidity documentation.

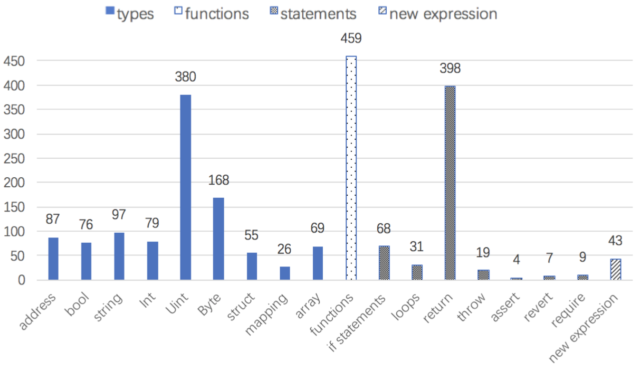

Figure 5. Number of Tests for Each Feature in the Solidity Compiler Test Set

From Table I, we can observe that the proposed Solidity semantics completely covers the supported high-level core features of the Solidity programming language. As for types, the semantics definition in the K-framework covers the following elementary types: `address`, `bool`, `string`, `Int`, `Uint` and `Byte`. `Fixed` and `Ufixed` are not covered because they are not fully supported by Solidity yet [46]. User-defined types, including `struct`, contract types and `enum`, are covered. Mappings, arrays, function types and `address payable` are also covered. The semantics associated with functions, such as function definitions and function calls, is fully covered. Furthermore, the semantics of statements is completely covered except that of `inline assembly statements` which are considered to be low-level features accessing EVM (i.e., this part of semantics can be integrated with KEVM [7]). All kinds of expressions in Solidity are covered. Lastly, the semantics of `event` is also covered. For all the parts of covered semantics, they are considered to be

Table I
COVERAGE OF THE PROPOSED SOLIDITY SEMANTICS

Features	**Coverage**	**Features**	**Coverage**	**Features**	**Coverage**
Types(Core)		**Statements(Core)**		**Expressions(Core)**	
Elementary Types		`If Statement`	FC	`Bitwise Operations`	FC
`address`	FC	`While Statement`	FC	`Arithmetic Operations`	FC
`bool`	FC	`For Statement`	FC	`Logical Operations`	FC
`string`	FC	`Block`	FC	`Comparison Operations`	FC
`Int`	FC	`Inline Assembly Statement`	N	`Assignment`	FC
`Uint`	FC			`Look Up`	FC
`Byte`	FC	`Do While Statement`	FC	`New Expression`	FC
`Fixed`	N	`Place Holder Statement`	FC	`Index Access`	FC
`Ufixed`	N	`Continue`	FC	`Member Access`	FC
User-defined Types	FC	`Break`	FC	`Other Expressions`	FC
Mappings	FC	`Return`	FC		
Array Types	FC	`Throw,Revert,Assert,Require`	FC	**Using For**	FC
Function Types	FC	`Simple Statement`	FC	**Inheritance**	FC
address payable	FC	`Emit Statement`	FC	**Event**	FC
Functions(Core)					
Function Definitions				*Function Calls*	
`Constructors`	FC	`Fallback Functions`	FC	`Internal Function Calls`	FC
`Normal Functions`	FC	`Modifiers`	FC	`External Function Calls`	FC

`FC`: Fully Covered and Consistent with Solidity IDE `N`: Not Covered

correct since the execution behaviours involved are consistent with the ones in the official Solidity compiler Remix. The parts of semantics for `using for` and `inheritance` are covered with rewriting. They are not the core features of Solidity since source programs with these features can be properly rewritten to equivalent forms, the semantics of which is completely supported. Therefore, the proposed Solidity semantics can be considered to be complete and correct in terms of the supported high-level core features of Solidity.

The semantics definition in the K-framework covers the semantics of the core features in smart contracts. Actually, the set of semantics in which known vulnerabilities of smart contracts lie has already been covered. Taking the DAO attack [3] as an example, the two vulnerabilities, reentrancy and call to the unknown [4], are mainly associated with the semantics of function calls. Thus, the proposed Solidity semantics can be used in the verification of smart contracts.

V. APPLICATIONS

In this section, we introduce the applications of the Solidity semantics as a formal foundation of the verification or security analysis of smart contracts. Particularly, we underline some practical problems in the existing approaches and show how these problems can be solved with the proposed semantics.

A. Formal Specification of Smart Contracts

The proposed Solidity semantics provides a formal specification of smart contracts written in Solidity due to the completeness and correctness of the semantics. Most of the existing tools, such as Oyente [6], Zeus [28], etc, are based on informal interpretations of the semantics of smart contracts. Specifically speaking, Oyente is based on a distilled EVM semantics which supports a subset of EVM features. As discussed in [11], the verification facilitated by Oyente is not sound due to the incompleteness of the EVM semantics and the patterns to detect vulnerabilities. Zeus interprets Solidity semantics with the semantics of an abstract language. Due to the nature of smart contract executions, the semantics of intermediate languages cannot be completely equivalent to that of Solidity in fundamental aspects which are important for the security analysis [19]. We show some examples below to illustrate the importance of a complete and correct Solidity semantics in the verification of smart contracts.

```
1  contract Bank {
2   mapping(address=>uint) credit;
3   address _BankLibrary;
4
5   constructor() public{
6     _BankLibrary = address(new BankLibrary());
7   }
8
9   function withdrawBalance(uint amount) public{
10    if(credit[msg.sender] >= amount){
11      _BankLibrary.delegatecall(abi.encodeWithSignature
12      ("withdraw(address payable, uint)", msg.sender,
13      amount));
14      credit[msg.sender] -= amount;
15    }
16  }
17 }
18
19 contract BankLibrary{
20  function withdraw(address payable recipient,
21  uint amount) public{
22    recipient.call.value(amount)(
23    abi.encodeWithSignature("nonExistingFunction()"));
24  }
25 }
```

Figure 6. Reentrancy False Negative in Oyente

A false negative in detecting delegated reentrancy attacks [14] in Oyente [6] is shown in Fig. 6. In this case, `delegatecall` is used to transfer ether rather than a direct `call`. After the ether has been transferred, the recipient's fallback function is invoked by `call` in lines $22-23$, making reentrancy attacks [4] possible. However, the semantics of `delegatecall` is not supported in Oyente, making it impossible to detect the reentrancy vulnerability associated with `delegatecall`. To be specific, Oyente detects the reen-

```
1  contract Bank {
2   uint credit = 100;
3
4   function withdraw(uint amount) public{
5     assert(amount <= 100 && credit > 0);
6     if(credit - amount >= 0){
7       credit = 0;
8       msg.sender.transfer(amount);
9     }
10  }
11 }
```

Figure 7. Integer Underflow False Positive in Zeus

trancy vulnerability by checking whether the path condition for executing CALL holds for updated state variables. If the ether transfer is achieved through delegatecall, it is impossible to obtain the path condition for executing the CALL instruction corresponding to the ether transfer in lines $22-23$ properly without the semantics of delegatecall. The detection of reentrancy attacks introduced by delegatecall is missing due to the incomplete EVM semantics. In addition, without the semantics of delegatecall, it is also impossible to detect the Parity wallet attack [56] in which the attacker exploited delegatecall to become the owner of the wallet contract to steal ether. Therefore, the completeness of the underlying semantics is important for the verification of smart contracts.

Apart from the completeness, the correctness of the semantics of smart contracts is crucial to the validity of the verification. For instance, Zeus [28] uses havoc statements to handle all state variables in the same way regardless of their types. havoc expands the domain of legitimate values that a state variable can take to the type-defined domain of that variable. In this case, initial values of state variables are ignored and the entire data domain is explored. This potentially leads to lots of false positives, especially in detecting integer underflow and overflow problems. An example of the false positives introduced by havoc is shown in Fig. 7.

In this example, the initial value of the state variable credit is 100, which requires that the amount of ether to be withdrawn for the first time cannot exceed 100 as indicated in the assertion in line 5. Before the ether is transferred, credit is set to be 0. However, havoc explores the entire data domain of credit regardless of its initial value, making it possible to trigger an integer underflow when credit is less than amount. This leads to a false positive in detecting integer underflow problems. The main reason for the false positive is that the semantics of havoc statements is not equivalent to that of state variable declarations in Solidity which takes into account initial values. Therefore, the correctness of the underlying semantics is crucial to the validity of the verification.

As illustrated above, both the completeness and correctness of the semantics of smart contracts are important in the verification and security analysis. Providing a complete and correct Solidity semantics solves the issues mentioned above as a formal specification of smart contracts. The proposed Solidity semantics allows us to refine the semantic foundations in the existing tools targeting Solidity, such as Zeus, etc, to improve their performance. Furthermore, the proposed semantics is not limited to the K-framework since it can be interpreted into a proof assistant language, such as Coq [57] or Isabelle [58], with $\mathbb{K}$ backends for formal reasoning [59].

B. Defining High-level Security Properties of Smart Contracts

The proposed Solidity semantics allows us to define high-level security properties of smart contracts for verification. Most of the existing tools, such as Oyente [6], Zeus [28], etc, focus on the detection of known vulnerabilities and attacks. This is limited since there can be vulnerabilities that have not been exploited to launch attacks. It is much more important to discover unknown vulnerabilities to prevent potential attacks. In addition, informal methods introduce false positives and negatives. Therefore, formal definitions of security properties which are not limited to specific attacks are necessary. In [11], a set of security properties is first defined for smart contracts on the proposed small-step EVM semantics. However, in addition to possible semantic gaps between high-level and low-level languages introduced by compiler bugs, the low-level definition of single-entrancy fails to address the high-level reentrancy property [60]. VerX [24] allows high-level specifications of temporal safety properties for low-level verification. However, high-level properties may not be precisely interpreted with EVM semantics due to its limitations. For instance, VerX integrates compiler behaviours into EVM level verification to make up for the loss of high-level semantic information during compilation. Securify [12] also defines a set of security patterns for analyzing smart contracts. However, pattern-based approaches introduce false positives without considering semantic-level correctness in the property specification. With the proposed semantics, it is possible to formally define semantic security properties at source code level, which facilitates the detection of unknown vulnerabilities and potential attacks and also excludes false positives introduced by pattern-based approaches. We show some examples below to illustrate the importance of defining high-level security properties of smart contracts.

A false positive in detecting the reentrancy vulnerability with single-entrancy based approaches (e.g., Oyente [6], Zeus [28], [11]) is shown in Fig. 8. In the withdraw function, the function nonExistingFunction() is invoked by call to transfer ether to msg.sender. If there is no function matching the specified function name in the recipient instance specified by msg.sender, the fallback function in that instance will be invoked, making it possible to re-enter the withdraw function with the recipient's fallback function. However, in this case the state variable credit is updated in line 6 before the execution of call in lines $7-9$, making the withdraw function reentrant and consequently excluding the possibility of reentrancy attacks. A function is said to be *reentrant* if it can be interrupted in the middle of its execution and then safely be called again ("re-entered") before its previous invocation's complete execution [60]. Single-entrancy based approaches to reentrancy attacks introduce false positives due to the lack of a formal definition of reentrancy.

```
contract Bank {
 mapping(address=>uint) credit;

 function withdraw(uint amount) public{
   if(credit[msg.sender] >= amount){
     credit[msg.sender] -= amount;
     msg.sender.call.value(amount)(
     abi.encodeWithSignature(
     "nonExistingFunction()"));
   }
 }
}
```

Figure 8. Reentrancy False Positive in Oyente and Zeus

```
1  contract Bank {
2   mapping(address=>uint) credit;
3
4   function withdraw(uint amount) public{
5     if(credit[msg.sender] >= amount){
6       credit[msg.sender] -= amount;
7       msg.sender.call.value(amount)(
8       abi.encodeWithSignature(
9       "nonExistingFunction()"));
10
11      credit[msg.sender] -= amount;
12      credit[msg.sender] += amount;
13    }
14  }
15 }
```

Figure 9. Reentrancy False Positive in Securify

A false positive in detecting the reentrancy vulnerability in Securify is shown in Fig. 9. Securify detects the reentrancy vulnerability by checking whether there is any instruction to write to the storage after any CALL instruction in any single trace. This is very close to detecting non-reentrant behaviours [60] in low-level instructions. However, this security pattern does not take into account semantic-level correctness. For instance, if we add two additional statements in lines $11-12$ to write to the storage after the call in lines $7-9$, this function is still reentrant since the value of credit at the end of the function is the same as the one before the call, which means that the withdraw function in Fig. 9 is equivalent to that in Fig. 8 at semantic level. In this case, a false positive is introduced. Therefore, defining high-level security properties for smart contracts based on semantic-level correctness is crucial to the performance of verification tools.

The formal definition of reentrancy presented below inherits the definition in [60] and excludes these false positives.

Notations. The runtime configuration of variables in a smart contract, denoted by $rc : G \cup L \mapsto D$, is a function mapping variables to their domain D at each execution step, where G and L are the sets of state variables and local variables, respectively. Given a basic block $\mathcal{B}$ with n statements, denoted by $\mathcal{B} \overset{\triangle}{=} S_1; S_2; \ldots; S_n$, we use rc_i to denote the runtime configuration after S_i is executed for $i \in \{1, 2, \ldots, n\}$, and rc_0 denotes the initial configuration before the block. We use $ExeEOS(rc, S)$ to denote the execution of a list of statements S on rc with the Solidity semantics. Then $rc_i = ExeEOS(rc_{i-1}, S_i) = ExeEOS(rc_0, (S_1; S_2; \ldots; S_i))$. Given two runtime configurations rc_i and rc_j both over $G \cup L$, we say $rc_i(X) = rc_j(X)$ where $X \subseteq G \cup L$, if 1) $X \neq \emptyset$ and $rc_i(x) = rc_j(x)$ for all $x \in X$ or 2) $X = \emptyset$. We use two particular state variables checkReturn and checkDelegate to record the logical conjunction of return values of unchecked low-level calls and delegatecalls in a smart contract, respectively. We use UI and SI to denote the sets of variables of unsigned and signed integer types where $UI \subseteq G \cup L$ and $SI \subseteq G \cup L$, respectively. T is a function mapping a variable to its type and Size is a function mapping an integer type to its size.

Reentrancy Safety. DAO attacks happen due to the fact that the associated function is not reentrant [60], making it possible to modify the values of critical state variables after an external call which may be achieved through delegatecall (e.g., Fig. 6). In Definition 1, a reentrant function requires that the values of critical state variables cannot be modified at semantic level after any external call. Theorem 1 shows that a smart contract is reentrant safe if every function in this contract is reentrant safe. For the runtime configuration calculation for this property, external calls are skipped to avoid the interference of another entry of the function.

Definition 1. *Given any function definition $D_F \overset{\triangle}{=} F(\vec{x})\{\ \mathcal{B}\ \}$ where $\mathcal{B} \overset{\triangle}{=} S_1; S_2; \ldots; S_n$ and a set of critical variables $C \subseteq G$, the function is called* reentrant *or* reentrant safe *if the following condition holds: if S_i is an external jumping statement (i.e., a statement with any external call or delegatecall) for some $i \in \{1, 2, \ldots, n\}$, then $rc_{i-1}(C) = rc_n(C)$.*

Theorem 1. *A smart contract is reentrant safe, if every function in this contract is reentrant safe.*

Proof. Let F_1 and F_2 be any two functions in the smart contract, and their statement sequences are denoted by $\sigma_1; \sigma_2; \ldots; \sigma_n$ and $\rho_1; \rho_2; \ldots; \rho_m$, respectively. If there is any reentrancy attack, we can always find a subsequence of the following form: $\sigma_1; \sigma_2; \ldots; \sigma_{i-1}; (\rho_1; \rho_2; \ldots; \rho_m); \sigma_{i+1}; \ldots; \sigma_n$ for some external jumping statement σ_i such that $rc_{\sigma_{i-1}}(C) \neq rc_{\sigma_n}(C)$. If F_1 and F_2 are reentrant, according to Definition 1, $rc_{\sigma_{i-1}}(C) = rc_{\sigma_n}(C)$. Therefore, if every function is reentrant, then such a subsequence does not exist. □

Based on Definition 1 and Theorem 1, Algorithm 1 is proposed to prove a smart contract is reentrant safe with the executable Solidity semantics. Given a smart contract, we try to prove that every function in this contract is reentrant. Different from some existing works (e.g., [28], [11], [6]) which use single-entrancy to avoid reentrancy attacks, this algorithm excludes non-reentrant behaviours [60] in every function of smart contracts. It also differs from trace-based approaches (e.g., [12], [30]) which rely on execution traces to verify certain properties of smart contracts since the definition of reentrancy is independent of any trace information. This benefits the static analysis of smart contracts.

Let us compare our approach with a trace-based detection approach to reentrancy attacks. The property of Effectively Callback Free (ECF) defined in [30] is based on execution

Algorithm 1: VerifyReentrancy(P)

```
input  : P: a smart contract
output: yes/no (to indicate whether the given smart
        contract is reentrant safe)
foreach function definition
  $D_F \triangleq F(\vec{x})\ \{\ S_1; S_2; \ldots; S_n\ \}$ do
    $rc_n \longleftarrow ExeEOS(rc_0, (S_1; S_2; \ldots; S_n))$;
    foreach statement $S_j$ do
        if $S_j$ is an external jumping statement and
          $rc_{j-1}(C) \neq rc_n(C)$ then return no;
        else
            $rc_j \longleftarrow ExeEOS(rc_{j-1}, S_j)$;
return yes;
```

traces which are regarded as a part of the semantics of contracts. This approach is effective in dynamic scenarios (e.g., online detection and dynamic verification) where trace information, i.e., a substantial part of the semantics, is available. However, it would be infeasible in automatic static analysis where no existing trace is given due to the lack of precise semantic foundations. Specifically speaking, this approach is facilitated by a simple imperative object-based programming language, called SMAC, the semantics of which is not completely equivalent to the Solidity semantics. For instance, this semantics does not support `delegatecall`, making it impossible to generate the traces of delegated reentrancy attacks (e.g., Fig. 6) properly. In the case of static analysis, as discussed above, the detection of delegated reentrancy attacks is missing. Compared with this approach, our approach is more powerful in static analysis due to the completeness and correctness of the Solidity semantics.

Also, the verification can be limited to a subset of state variables to be more specific to certain reentrancy attacks, which is impossible on EVM semantics for compound types. If this security property is specific to the test set, there will be no false positives and negatives in the verification. With respect to the test cases shown in Fig. 8 and Fig. 9, the reentrancy property defined in Definition 1 is specific when $C = G$ so that these two cases can be correctly identified as secure contracts in terms of reentrancy. We implement Algorithm 1 in $\mathbb{K}$ backends and verify four variants of real DAO contracts obtained from [61] in which the reentrancy bug has been fixed (i.e., state variables are updated before ether transfer in the `withdraw` function). These four real contracts are verified to be reentrant safe. Oyente and Zeus cannot identify them correctly due to the reasons mentioned above.

Delegatecall Safety. `delegatecall` is executed in the context of the caller rather than the recipient, making it possible to modify the caller state. This introduces certain security vulnerabilities. For instance, `delegatecall` was exploited in the Parity wallet attack [56] which led to a huge impact on digital assets [62]. Delegatecall safety requires that the critical state of the caller cannot be modified at semantic level through `delegatecall` and `delegatecall` must return true. The first condition guarantees that critical state variables, such as the owner of the contract, cannot be modified through `delegatecall`, and the second condition requires that `delegatecall` must be successful in transferring ether to recipients to avoid frozen ether.

Definition 2. *Given any function definition* $D_F \triangleq F(\vec{x})\{\ \mathcal{B}\ \}$ *where* $\mathcal{B} \triangleq S_1; S_2; \ldots; S_n$*, an initial state* $rc_0(checkDelegate) = true$ *and a set of critical variables* $C \subseteq G$*, the function is called* delegatecall safe *if the following condition holds: if* S_i *is a delegatecall statement (i.e., a statement with any delegatecall) for some* $i \in \{1, 2, \ldots, n\}$*, then* $rc_{i-1}(C) = rc_i(C)$ *and* $rc_i(checkDelegate) = true$*.*

Theorem 2. *A smart contract is delegatecall safe, if every function in this contract is delegatecall safe.*

Proof. Similar to Theorem 1. □

Exception Handling Correctness. Exception disorders are caused by the absent checking of return values of low-level calls, such as `call`, `send`, etc. If such low-level calls fail, exceptions will not be propagated to outer calls. Exception handling correctness requires that low-level calls without exception handling, such as assertion checking of return values, must succeed to avoid exception disorders.

Definition 3. *Given any function definition* $D_F \triangleq F(\vec{x})\{\ \mathcal{B}\ \}$ *where* $\mathcal{B} \triangleq S_1; S_2; \ldots; S_n$ *and an initial state* $rc_0(checkReturn) = true$*, the function is called* exception handling correct *if the following condition holds: if* S_i *is an unchecked low-level jumping statement (i.e., a statement with any unchecked low-level call) for some* $i \in \{1, 2, \ldots, n\}$*, then* $rc_i(checkReturn) = true$*.*

Theorem 3. *A smart contract is exception handling correct, if every function in this contract is exception handling correct.*

Proof. Similar to Theorem 1. □

Integer Arithmetic Operation Correctness. This security property excludes integer overflow and underflow problems.

Definition 4. *Given any function definition* $D_F \triangleq F(\vec{x})\{\ \mathcal{B}\ \}$ *where* $\mathcal{B} \triangleq S_1; S_2; \ldots; S_n$*, the function is called* integer arithmetic operation correct *if the following condition holds: if* S_i *is an integer arithmetic operation statement (i.e., a statement with any integer arithmetic operation) for some* $i \in \{1, 2, \ldots, n\}$*, then* $0 <= rc_i(x) < 2^{Size(T(x))}$ *for all* $x \in UI$ *and -* $2^{Size(T(x))-1} <= rc_i(x) < 2^{Size(T(x))-1}$ *for all* $x \in SI$*.*

Theorem 4. *A smart contract is integer arithmetic operation correct, if every function in this contract is integer arithmetic operation correct.*

Proof. There are two cases in constructing the proof. 1) If there is no function call in any function of the contract, then integer arithmetic operation correctness in one function is

```
contract Test {
 uint256 a = 1;
 uint256[2] b = [1,2];

 function foo() public {
   uint256[2] d;
   d[0] = 7;
   d[1] = 8;
 }
}
```

Figure 10. A Bug in the Remix Compiler before Version 0.5.0

independent of the other functions. In this case, Theorem 4 holds. 2) If there exists at least one function call in any function of this contract, then the proof is constructed in a way similar to Theorem 1. □

For delegatecall safety, exception handling correctness and integer arithmetic operation correctness, the verification algorithms are constructed in a way similar to Algorithm 1. Due to limit of space, they are omitted.

C. Defining Correct and Secure High-level Execution Behaviours of Smart Contracts

As mentioned above, the execution behaviours of compilers may be inconsistent with the high-level Solidity semantics so that low-level bytecode may not equivalently capture intended high-level execution behaviours. Another application of the proposed Solidity semantics is defining correct and secure high-level execution behaviours of smart contracts to reason about Solidity compiler bugs and assist developers in writing secure smart contracts in Solidity.

1) Reasoning about Compiler Bugs: A bug in the earlier versions of Remix [47] is shown in Fig. 10. This smart contract can be compiled in the Remix compiler before Version 0.5.0. Please note that the local array declared in the function `foo`, named `d`, is not specified with any location information (i.e., `storage` or `memory`). In this case, the values of `d` overwrite the ones in the storage from the first slot. Specifically speaking, the values of the first and the second indexes of `d` overwrite the values of `a` and the first index of b, respectively. However, this is not consistent with the intention of developers who expect that the allocation of `d` takes place in the memory. A correct semantics of a local array declaration requires that the location information of this array must be specified. If it is specified with `memory`, the allocation takes place in the local memory. If it is specified with `storage`, a pointer to the storage must be provided. In other cases, the local array declaration is considered to be invalid in syntax.

Although this bug has been fixed in Remix since Version 0.5.0, the proposed semantics allows us to discover other compiler bugs through the equivalence checking of the Solidity semantics and compilers which is conducted by automatically comparing the execution behaviours of Solidity contracts on the proposed semantics with the ones of the compiled bytecode on EVM semantics [7] in the K-framework [49].

2) Assisting Developers in Writing Secure Smart Contracts: As illustrated above, excluding invalid syntax and undefined behaviours with the proposed Solidity semantics is a good way to assist developers in writing secure smart contracts. In addition, correct and secure high-level execution behaviours are defined in the semantics. For instance, according to the low-level implementations of the language features of Solidity [46], exceptions are not propagated by low-level calls, such as `call`, `send`, etc, resulting in exception disorders. However, this is not the intention of developers who expect that exceptions in all kinds of calls propagate properly from the perspective of source code. Instead of being consistent with the low-level implementations, a uniform exception handling mechanism is applied in all kinds of calls to propagate exceptions in a proper way, which excludes exception disorders at high level. At the same time, exception disorders in the low-level implementations can also be detected with the semantics by checking return values of low-level calls. In this way, the proposed semantics assists developers in writing secure smart contracts by checking whether the high-level semantics is consistent with the low-level implementations. If any inconsistency occurs, developers are advised to use alternative expressions. As mentioned above, this equivalence checking procedure can be automatically conducted in the K-framework with the proposed Solidity semantics and KEVM [7].

The benefits of defining correct and secure high-level execution behaviours of smart contracts instead of being consistent with the low-level implementations lie in two aspects. First, as illustrated above, the low-level implementations are not necessarily correct and there can be compiler bugs. Secondly, alternative low-level implementations [63] have been proposed for smart contracts. Sticking to the existing ones may affect the longstanding performance of the high-level Solidity semantics.

VI. Conclusion

In this paper, we introduce an executable operational semantics of Solidity formalized in the K-framework. We present the semantics of the core features of Solidity, namely memory operations, new contract instance creations and function calls, with an emphasis on the semantics of the exception handling features. The semantics is designed from a general point of view to adapt to the language evolution of smart contracts. Experiment results show that the proposed Solidity semantics has already completely covered the supported high-level core language features specified by the official Solidity documentation, and the covered semantics is consistent with the official Solidity compiler Remix. Furthermore, the applications of the proposed Solidity semantics in the verification and security analysis of smart contracts are discussed.

Acknowledgments

This work is supported by the Ministry of Education, Singapore under its Tier-2 Project (Award Number: MOE2018-T2-1-068) and partially supported by the National Research Foundation, Singapore under its NSoE Programme (Award Number: NSOE-TSS2019-03).

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[2] K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *FC*, 2016, pp. 79–94.

[3] D. Siegel. (2016) Understanding the DAO attack. [Online]. Available: http://www.coindesk.com/understanding-dao-hack-journalists

[4] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *POST 2017*, ser. Lecture Notes in Computer Science, M. Maffei and M. Ryan, Eds., vol. 10204. Springer, 2017, pp. 164–186.

[5] A. Madeira. (2019) The DAO, the hack, the soft fork and the hard fork. [Online]. Available: https://www.cryptocompare.com/coins/guides/the-dao-the-hack-the-soft-fork-and-the-hard-fork

[6] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *CCS 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 254–269.

[7] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. M. Moore, D. Park, Y. Zhang, A. Ştefănescu, and G. Roşu, "KEVM: A complete formal semantics of the Ethereum Virtual Machine," in *CSF 2018*. IEEE Computer Society, 2018, pp. 204–217.

[8] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *ASE 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 259–269.

[9] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying Ethereum smart contract bytecode in Isabelle/HOL," in *CPP 2018*. ACM, 2018, pp. 66–77.

[10] Y. Hirai, "Defining the Ethereum Virtual Machine for interactive theorem provers," in *FC*, M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, Eds. Cham: Springer International Publishing, 2017, pp. 520–535.

[11] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of Ethereum smart contracts," in *POST 2018*, pp. 243–269.

[12] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts," in *CCS 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 67–82.

[13] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving out-of-gas conditions in Ethereum smart contracts," *PACMPL*, vol. 2, no. OOPSLA, pp. 116:1–116:27, 2018.

[14] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *NDSS 2019*. The Internet Society, 2019.

[15] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *ACSAC 2018*. ACM, 2018, pp. 653–663.

[16] Mythril. [Online]. Available: https://github.com/ConsenSys/mythril

[17] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *ASE 2019*. IEEE, 2019, pp. 1186–1189.

[18] J. Krupp and C. Rossow, "teEther: Gnawing at Ethereum to automatically exploit smart contracts," in *USENIX 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 1317–1333.

[19] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of Ethereum smart contracts," in *CAV 2018*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 51–78.

[20] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," in *ISSTA 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 363–373.

[21] S. Wang, C. Zhang, and Z. Su, "Detecting nondeterministic payment bugs in Ethereum smart contracts," *OOPSLA*, pp. 189:1–189:29, 2019.

[22] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "TokenScope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in Ethereum," in *CCS 2019*, pp. 1503–1520.

[23] J. He, M. Balunovic, N. Ambroladze, P. Tsankov, and M. T. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *CCS 2019*, pp. 531–548.

[24] VerX: Safety verification of smart contracts. [Online]. Available: https://verx.ch

[25] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "sCompile: Critical path identification and analysis for smart contracts," in *ICFEM 2019*, pp. 286–304.

[26] H. Wang, Y. Li, S. Lin, L. Ma, and Y. Liu, "VULTRON: Catching vulnerable smart contracts once and for all," in *ICSE (NIER) 2019*, A. Sarma and L. Murta, Eds. IEEE / ACM, 2019, pp. 1–4.

[27] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Z. Béguelin, "Formal verification of smart contracts," in *PLAS@CCS 2016*, T. C. Murray and D. Stefan, Eds. ACM, 2016, pp. 91–96.

[28] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *NDSS 2018*. The Internet Society, 2018.

[29] S. K. Lahiri, S. Chen, Y. Wang, and I. Dillig, "Formal specification and verification of smart contracts for Azure Blockchain," *arXiv preprint*, vol. abs/1812.08829, 2018.

[30] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *POPL*, vol. 48:1–48:28, 2018.

[31] A. Mavridou and A. Laszka, "FSolidM for designing secure Ethereum smart contracts," in *POST 2018*, pp. 270–277.

[32] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of Ethereum smart contracts," in *WETSEB@ICSE 2018*. ACM, 2018, pp. 9–16.

[33] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *WETSEB@ICSE 2019*. IEEE / ACM, 2019, pp. 8–15.

[34] Z. Nehai and F. Bobot, "Deductive proof of Ethereum smart contracts using Why3," *arXiv preprint*, vol. abs/1904.11281, 2019.

[35] Á. Hajdu and D. Jovanovic, "solc-verify: A modular verifier for Solidity smart contracts," *arXiv preprint*, vol. abs/1907.04262, 2019.

[36] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VeriSmart: A highly precise safety verifier for Ethereum smart contracts," *arXiv preprint*, vol. abs/1908.11227, 2019.

[37] X. Li, Z. Shi, Q. Zhang, G. Wang, Y. Guan, and N. Han, "Towards verifying Ethereum smart contracts at intermediate language level," in *ICFEM 2019*, pp. 121–137.

[38] Vyper documentation. [Online]. Available: https://vyper.readthedocs.io/en/latest

[39] Bamboo. [Online]. Available: https://github.com/pirapira/bamboo

[40] Flint. [Online]. Available: https://github.com/flintlang/flint

[41] Etherscan. [Online]. Available: https://etherscan.io

[42] Z. Yang and H. Lei, "Lolisa: Formal syntax and semantics for a subset of the Solidity programming language," *arXiv preprint*, vol. abs/1803.09885, 2018.

[43] J. Zakrzewski, "Towards verification of Ethereum smart contracts: A formalization of core of Solidity," in *VSTTE 2018*, ser. Lecture Notes in Computer Science, vol. 11294. Springer, 2018, pp. 229–247.

[44] S. Crafa, M. Pirro, and E. Zucca, "Is Solidity solid enough?" in *FC*, 2019.

[45] M. Bartoletti, L. Galletta, and M. Murgia, "A minimal core calculus for Solidity contracts," in *DPM/CBT@ESORICS*, 2019.

[46] Solidity documentation. [Online]. Available: https://solidity.readthedocs.io/en/latest

[47] Remix - Solidity IDE. [Online]. Available: https://remix-ide.readthedocs.io/en/latest

[48] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols," ser. IFIP, B. Preneel, Ed., vol. 152. Kluwer, 1999, pp. 258–272.

[49] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[50] N. Martí-Oliet and J. Meseguer, "Rewriting logic: Roadmap and bibliography," *Theor. Comput. Sci.*, vol. 285, pp. 121–154, 2002.

[51] D. Bogdănaş and G. Roşu, "K-Java: A complete semantics of Java," in *POPL 2015*. ACM, 2015, pp. 445–456.

[52] C. Ellison and G. Roşu, "An executable formal semantics of C with applications," in *POPL 2012*. ACM, 2012, pp. 533–544.

[53] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-based program verifiers for all languages," in *OOPSLA 2016*, E. Visser and Y. Smaragdakis, Eds. ACM, 2016, pp. 74–91.

[54] T. Nipkow and G. Klein, "IMP: A simple imperative language," *Concrete Semantics. Springer, Cham*, 2014.

[55] Solidity compiler test set. [Online]. Available: https://github.com/ethereum/solidity

[56] S. Palladino. The Parity wallet attack. [Online]. Available: https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7

[57] The Coq proof assistant. [Online]. Available: http://coq.inria.fr

[58] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, 2002.

[59] K-framework. [Online]. Available: http://www.kframework.org/index.php

[60] M. Kerrisk, *The Linux Programming Interface*. No Starch Press, 2010.

[61] The standard DAO framework. [Online]. Available: https://github.com/slockit/DAO

[62] M. Suiche. (2017) The $280M Ethereum's Parity bug. [Online]. Available: https://blog.comae.io/the-280m-ethereums-bug-f28e5de43513

[63] T. Kasampalis, D. Guth, B. Moore, T. Şerbănuţă, V. Şerbănuţă, D. Filaretti, G. Roşu, and R. Johnson, "IELE: An intermediate-level blockchain language designed and implemented using formal semantics," July 2018.

Appendix

A. ReadAddress

We evaluate global and local variables with ReadAddress-GlobalVariables and ReadAddress-LocalVariables, respectively. Particularly, global and local variables are stored in `ctStorage` and `Memory` in the corresponding instance, respectively. The gas consumption for reading a slot is calculated and updated in `gasCal`.

B. WriteAddress

We rewrite global variables with WriteAddress-GlobalVariables and local variables with WriteAddress-LocalVariables. Particularly, global and local variables are rewritten in `ctStorage` and `Memory` in the corresponding instance, respectively. The gas consumption for rewriting a slot is calculated and updated in `gasCal`.

C. AllocateAddress

A memory slot is allocated for a variable through `allocateAddress`. First, we map the corresponding contract instance with its Id `N`. Then we add a new slot at `Addr` with the initial value `V`. Particularly, for global variables the slot is added in `ctStorage` as shown in AllocateAddress-GlobalVariables. And for local variables, the slot is added in `Memory` as illustrated in AllocateAddress-LocalVariables. The gas consumption for allocating a slot is calculated and updated in `gasCal`.

D. UpdateState

The blockchain state is updated through `updateState` when a new contract instance is created. To be specific, the number of contract instances is increased by 1 in `cntContracts`. A new contract instance cell is created with its Id `N` in `ctId` and the associated contract name `X` in `ctName`. In addition, as shown in UpdateState-Main-Contract, if this new contract instance creation is in the "Main" contract, the number of transactions will be increased by 1 in `cntTrans` and a new mapping will be created to record this new contract instance creation as a transaction in `tranComputation`. As shown in UpdateState-Function-Call, if this new contract instance creation is nested in a function call, no transaction information will be recorded since it is not an independent transaction. Finally, a new list item of `X` is added into `newStack` to indicate that we are in the process of a new contract instance creation. The gas consumption for deploying a new contract instance is calculated and updated in `gasCal(#newInstance)`.

RULE ReadAddress-GlobalVariables

$$\left\langle \frac{\texttt{readAddress(Addr:Int,String2Id("Global"))}}{\texttt{gasCal(\#read,String2Id("Global"))} \curvearrowright \texttt{V:Value}} \cdots \right\rangle_{k}$$
$$\left\langle \texttt{ListItem(N:Int)} \cdots \right\rangle_{contractStack}$$
$$\left\langle \begin{array}{c} \langle \;\texttt{N}\; \rangle_{ctId} \\ \langle \cdots \;\texttt{Addr |-> V}\; \cdots \rangle_{ctStorage} \end{array} \cdots \right\rangle_{contractInstance}$$

RULE ReadAddress-LocalVariables

$$\left\langle \frac{\texttt{readAddress(Addr:Int,String2Id("Local"))}}{\texttt{gasCal(\#read,String2Id("Local"))} \curvearrowright \texttt{V:Value}} \cdots \right\rangle_{k}$$
$$\left\langle \texttt{ListItem(N:Int)} \cdots \right\rangle_{contractStack}$$
$$\left\langle \begin{array}{c} \langle \;\texttt{N}\; \rangle_{ctId} \\ \langle \cdots \;\texttt{Addr |-> V}\; \cdots \rangle_{Memory} \end{array} \cdots \right\rangle_{contractInstance}$$

RULE WriteAddress-GlobalVariables

$$\left\langle \frac{\begin{array}{c}\texttt{writeAddress(Addr:Int,String2Id("Global"),} \\ \texttt{V:Value)}\end{array}}{\texttt{gasCal(\#write,String2Id("Global"),OV,V)} \curvearrowright \texttt{V}} \cdots \right\rangle_{k}$$
$$\left\langle \texttt{ListItem(N:Int)} \cdots \right\rangle_{contractStack}$$
$$\left\langle \begin{array}{c} \langle \;\texttt{N}\; \rangle_{ctId} \\ \left\langle \cdots \dfrac{\texttt{Addr |-> OV}}{\texttt{Addr |-> V}} \cdots \right\rangle_{ctStorage} \end{array} \cdots \right\rangle_{contractInstance}$$

RULE WriteAddress-LocalVariables

$$\left\langle \frac{\begin{array}{c}\texttt{writeAddress(Addr:Int,String2Id("Local"),} \\ \texttt{V:Value)}\end{array}}{\texttt{gasCal(\#write,String2Id("Local"),OV,V)} \curvearrowright \texttt{V}} \cdots \right\rangle_{k}$$
$$\left\langle \texttt{ListItem(N:Int)} \cdots \right\rangle_{contractStack}$$
$$\left\langle \begin{array}{c} \langle \;\texttt{N}\; \rangle_{ctId} \\ \left\langle \cdots \dfrac{\texttt{Addr |-> OV}}{\texttt{Addr |-> V}} \cdots \right\rangle_{Memory} \end{array} \cdots \right\rangle_{contractInstance}$$

RULE AllocateAddress-GlobalVariables

$$\left\langle \frac{\begin{array}{c}\texttt{allocateAddress(N:Int, Addr:Int,} \\ \texttt{String2Id("Global"), V:Value)}\end{array}}{\texttt{gasCal(\#allocate,String2Id("Global"))} \curvearrowright \texttt{V}} \cdots \right\rangle_{k}$$
$$\left\langle \begin{array}{c} \langle \;\texttt{N}\; \rangle_{ctId} \\ \left\langle \dfrac{\texttt{STORAGE:Map}}{\texttt{STORAGE (Addr |-> V)}} \right\rangle_{ctStorage} \end{array} \cdots \right\rangle_{contractInstance}$$

RULE AllocateAddress-LocalVariables

$$\left\langle \frac{\begin{array}{c}\texttt{allocateAddress(N:Int, Addr:Int,} \\ \texttt{String2Id("Local"), V:Value)}\end{array}}{\texttt{gasCal(\#allocate,String2Id("Local"))} \curvearrowright \texttt{V}} \cdots \right\rangle_{k}$$
$$\left\langle \begin{array}{c} \langle \;\texttt{N}\; \rangle_{ctId} \\ \left\langle \dfrac{\texttt{MEMORY:Map}}{\texttt{MEMORY (Addr |-> V)}} \right\rangle_{Memory} \end{array} \cdots \right\rangle_{contractInstance}$$

E. AllocateStorage

Memory slots are allocated for state variables in the new contract instance through `allocateStateVars`. As shown in AllocateStateVariables, a memory slot is allocated for each variable `Var` with `allocate` sequentially until there are no more variables to process.

F. InitInstance

If there is a constructor in the contract for which a new instance is created, InitInstance-WithConstructor will be applied. Otherwise, InitInstance-NoConstructor will be applied. The rule InitInstance-NoConstructor simply returns the Id of the new instance. Furthermore, the associated contract name `X` is popped out of `newStack` to

RULE UpdateState-Main-Contract

$$\left\langle \frac{\texttt{updateState(X:Id)}}{\texttt{gasCal(\#newInstance)}}\cdots\right\rangle_{k} \left\langle \left\langle \texttt{X} \right\rangle_{cName} \cdots\right\rangle_{contract}$$

$$\left\langle \frac{\texttt{N:Int}}{\texttt{N +Int 1}}\right\rangle_{cntContracts} \left\langle \frac{\texttt{T:Int}}{\texttt{T +Int 1}}\right\rangle_{cntTrans}$$

$$\left\langle \frac{\texttt{INS:Bag}}{\texttt{INS} \left\langle \begin{array}{c} \left\langle \texttt{N} \right\rangle_{ctId} \\ \left\langle \texttt{X} \right\rangle_{ctName} \end{array} \cdots\right\rangle_{contractInstance}}\right\rangle_{contractInstances}$$

$$\left\langle \frac{\texttt{Trans:Map}}{\texttt{Trans (T |-> "new contract")}}\right\rangle_{tranComputation}$$

$$\left\langle \frac{\texttt{L:List}}{\texttt{ListItem(X) L}}\right\rangle_{newStack} \left\langle \texttt{.List} \right\rangle_{functionStack}$$

RULE UpdateState-Function-Call

$$\left\langle \frac{\texttt{updateState(X:Id)}}{\texttt{gasCal(\#newInstance)}}\cdots\right\rangle_{k} \left\langle \left\langle \texttt{X} \right\rangle_{cName} \cdots\right\rangle_{contract}$$

$$\left\langle \frac{\texttt{N:Int}}{\texttt{N +Int 1}}\right\rangle_{cntContracts}$$

$$\left\langle \frac{\texttt{INS:Bag}}{\texttt{INS} \left\langle \begin{array}{c} \left\langle \texttt{N} \right\rangle_{ctId} \\ \left\langle \texttt{X} \right\rangle_{ctName} \end{array} \cdots\right\rangle_{contractInstance}}\right\rangle_{contractInstances}$$

$$\left\langle \frac{\texttt{L:List}}{\texttt{ListItem(X) L}}\right\rangle_{newStack} \left\langle \texttt{CallList:List} \right\rangle_{functionStack}$$

`requires CallList =/=K .List`

RULE AllocateStorage

$$\left\langle \frac{\texttt{allocateStorage(X:Id)}}{\texttt{allocateStateVars(N -Int 1, Vars)}}\cdots\right\rangle_{k}$$

$$\left\langle \left\langle \texttt{X} \right\rangle_{cName} \left\langle \texttt{Vars:List} \right\rangle_{stateVars} \cdots\right\rangle_{contract}$$

$$\left\langle \texttt{N:Int} \right\rangle_{cntContracts}$$

RULE AllocateStateVariables

$$\left\langle \frac{\begin{array}{l}\texttt{allocateStateVars(N:Int,} \\ \texttt{ ListItem(Var) Vars:List)}\end{array}}{\begin{array}{l}\texttt{ allocate(N, Var)} \curvearrowright \\ \texttt{allocateStateVars(N, Vars)}\end{array}}\cdots\right\rangle_{k}$$

RULE AllocateStateVariables-End

$$\left\langle \frac{\texttt{allocateStateVars(N:Int, .List)}}{\cdot}\cdots\right\rangle_{k}$$

RULE InitInstance-NoConstructor

$$\left\langle \frac{\texttt{initInstance(X:Id,E:ExpressionList)}}{\texttt{N -Int 1}}\cdots\right\rangle_{k}$$

$$\left\langle \frac{\texttt{ListItem(X) L:List}}{\texttt{L}}\right\rangle_{newStack} \left\langle \texttt{N:Int} \right\rangle_{cntContracts}$$

$$\left\langle \left\langle \texttt{X} \right\rangle_{cName} \left\langle \texttt{false} \right\rangle_{Constructor} \cdots\right\rangle_{contract}$$

RULE InitInstance-WithConstructor

$$\left\langle \frac{\texttt{initInstance(X:Id,E:ExpressionList)}}{\begin{array}{c}\texttt{functionCall(C;N -Int 1;} \\ \texttt{String2Id("constructor");E;} \\ \texttt{\#msgInfo(C,N -Int 1,0,gasCal(\#constructor)))}\end{array}}\cdots\right\rangle_{k}$$

$$\left\langle \frac{\texttt{ListItem(X) L:List}}{\texttt{L}}\right\rangle_{newStack} \left\langle \texttt{N:Int} \right\rangle_{cntContracts}$$

$$\left\langle \texttt{ListItem(C:Int)} \cdots\right\rangle_{contractStack}$$

$$\left\langle \left\langle \texttt{X} \right\rangle_{cName} \left\langle \texttt{true} \right\rangle_{Constructor} \cdots\right\rangle_{contract}$$

indicate that the new instance creation is finished. While in the rule InitInstance-WithConstructor, apart from removing the contract name `X` out of `newStack`, a function call is processed to execute the constructor. To be specific, the caller of this function is `C` and the recipient is the new instance `N - 1`. In addition, the function to be called is the constructor and `E` specifies the function arguments. The name of the constructor has been changed to “constructor” since Version 0.4.22 [46]. The function name is stored as an identifier, so we transform the string “constructor” into the equivalent `Id` type with the built-in function `String2Id` in the K-framework. The last argument of `functionCall` is the “msg” information, denoted by `#msgInfo(msg.sender(Id_of_Caller), Id_of_Recipient, msg.value, msg.gas)`. The gas consumption for executing the constructor is calculated and updated in `gasCal(#constructor)`.

G. *SwitchContext*

RULE Switch-Context

$$\left\langle \frac{\texttt{switchContext(C:Int,R:Int,F:Id,M:Msg)}}{\texttt{createTransaction(L)}}\cdots\right\rangle_{k}$$

$$\left\langle \frac{\texttt{L:List}}{\texttt{ListItem(R) L}}\right\rangle_{contractStack} \left\langle \texttt{CNum} \right\rangle_{cntContracts}$$

$$\left\langle \frac{\texttt{M1}}{\texttt{M}}\right\rangle_{Msg} \left\langle \frac{\texttt{MsgList:List}}{\texttt{ListItem(M1) MsgList}}\right\rangle_{msgStack}$$

$$\left\langle \frac{\texttt{CallList:List}}{\begin{array}{c}\texttt{ListItem(\#state(RhoC,F,} \\ \texttt{\#return(false,0),CNum,false)) CallList}\end{array}}\right\rangle_{functionStack}$$

$$\left\langle \begin{array}{l} \left\langle \texttt{C} \right\rangle_{ctId} \left\langle \texttt{RhoG} \right\rangle_{globalContext} \\ \qquad \left\langle \frac{\texttt{RhoC}}{\texttt{RhoG}}\right\rangle_{ctContext} \end{array} \cdots\right\rangle_{contractInstance}$$

$$\left\langle \frac{\texttt{G}}{\texttt{0}}\right\rangle_{gasConsumption} \left\langle \frac{\texttt{GasList:List}}{\texttt{ListItem(G) GasList}}\right\rangle_{gasStack}$$

The first step for a function call is to switch to the recipient instance as shown in Switch-Context. This is achieved by adding the recipient `R` into `contractStack` which indicates the current contract instance. At the same time, we need to store the state information of this function call, presented as `#state(RhoC,F,#return(false,0),CNum,false)`, in `functionStack`. There are altogether five items in the state information. The first item `RhoC` is the variable context of the caller instance which can be used to read and write variables in that instance. The second one `F` is the name of the function to be called. The next one is the information for `return` with two fields. The first field indicates whether a return statement has already been encountered, while the second records the return value. We assume that the default return value is 0. After this, the next item in `#state` is the number of contract instances created before this function call which is `CNum`. The last item is a flag to indicate whether this function call throws an exception.

The variable context of the caller instance `RhoC` is obtained from the cell `contractInstance` with the caller instance Id `C` in the sub-cell `ctId`. Apart from storing the previous variable context in `functionStack`, we rewrite it to `RhoG` which is stored in `globalContext` and only associated with state variables. The intention of this step is to remove the context of local variables. Furthermore, the current transaction information in `Msg` is rewritten to `M`, while the previous one `M1` is pushed into `msgStack`. Similarly, the current gas consumption in `gasConsumption` is rewritten to 0, and the previous one `G` is pushed into `gasStack`. Finally, we record the transaction information through `createTransaction`. Particularly, the balance of the creator of the transaction is reduced at a certain rate to pay for the gas in this sub-step. Due to limit of space, this part is omitted.

RULE INTERNAL-FUNCTION-CALL

$$\begin{array}{l}\left\langle \dfrac{\texttt{functionCall(F:Id;Es:Values)}}{\begin{array}{c}\texttt{saveCurContext(CNum,0)} \curvearrowright\\ \texttt{call(searchFunction(F,checkCallData(Es,0)),Es)}\\ \curvearrowright \texttt{updateCurContext(CNum,0)}\end{array}} \cdots \right\rangle_{k}\\ \left\langle \texttt{CNum} \right\rangle_{cntContracts}\end{array}$$

RULE CALL

$$\left\langle \dfrac{\texttt{call(N:Int,Es:Values)}}{\begin{array}{c}\texttt{initFunParams(N,Es)} \curvearrowright\\ \texttt{processFunQuantifiers(N)}\curvearrowright\\ \texttt{callFunBody(N)}\end{array}} \cdots \right\rangle_{k}$$

RULE CALL-FUNCTION-BODY

$$\begin{array}{l}\left\langle \dfrac{\texttt{callFunBody(N)}}{\begin{array}{c}\texttt{funBody(B)} \curvearrowright \texttt{updateReturnParams(N)} \curvearrowright\\ \texttt{updateReturnValue(N)}\end{array}} \cdots \right\rangle_{k}\\ \left\langle \left\langle \mathrm{N} \right\rangle_{fId} \left\langle \mathrm{B} \right\rangle_{Body} \cdots \right\rangle_{function}\end{array}$$

RULE FUNCTION-BODY

$$\left\langle \dfrac{\texttt{funBody(S:Statement Ss:Statements)}}{\texttt{exeStmt(S)} \curvearrowright \texttt{funBody(Ss)}} \cdots \right\rangle_{k}$$

$$\left\langle \dfrac{\texttt{funBody(.Statements)}}{\cdot} \cdots \right\rangle_{k}$$

H. Internal-Function-Call

There are three sub-steps in handling internal function calls. To be specific, saveCurContext facilitates the semantics of exceptions. When a transaction throws an exception, the states of all the contract instances involved should revert to the ones before this transaction. The instance states prior to the transaction are saved through saveCurContext(CNum,0) where CNum is the number of contract instances created before this function call and 0 specifies the starting point. In other words, we store the states of contract instances whose Ids range from 0 to CNum - 1. If this is a nested call, the states of the involving instances will not be saved through saveCurContext since it aims to keep track of the states before a transaction. call(searchFunction(F,checkCallData(Es,0)), Es) is the actual call of the function F with arguments Es. Particularly, searchFunction is an expression that returns the Id of the function to be called. It is used to distinguish functions with the same name F through checkCallData which checks the call data specified by Es. The second argument of checkCallData records the number of parameters that have been checked, so we start from 0. Finally, we update the instance states we have saved to the ones after the function call through updateCurContext(CNum,0). If an exception is encountered in this call, the states of the involving instances will not be updated through updateCurContext.

In dealing with CALL, we first initialize function parameters including input parameters and return parameters through initFunParams. Input parameters are initialized by the function call arguments Es and return parameters are initialized to be the default values, such as 0 for an integer and false for a Boolean type. The first argument of initFunParams, denoted by N, is the Id of the function to be called while the second Es specifies the values of the input parameters of the function. After this, processFunQuantifiers deals with

RULE RETURN-CONTEXT

$$\begin{array}{l}\left\langle \dfrac{\texttt{returnContext(R:Int)}}{\begin{array}{c}\texttt{clearRecipientContext(R,RhoG)}\curvearrowright\\ \texttt{clearCallerContext(C,Rho)}\curvearrowright\\ \texttt{propagateException(C,Exception)}\curvearrowright\texttt{E:Value}\end{array}} \cdots \right\rangle_{k}\\ \left\langle \dfrac{\texttt{ListItem(R) ListItem(C) L:List}}{\texttt{ListItem(C) L}} \right\rangle_{contractStack}\\ \left\langle \dfrac{\texttt{M}}{\texttt{M1}} \right\rangle_{Msg} \left\langle \dfrac{\texttt{ListItem(M1) MsgList:List}}{\texttt{MsgList}} \right\rangle_{msgStack}\\ \left\langle \dfrac{\begin{array}{c}\texttt{ListItem(\#state(Rho,\_,\#return(\_,E),}\\ \texttt{\_,Exception)) CallList:List}\end{array}}{\texttt{CallList}} \right\rangle_{functionStack}\\ \left\langle \left\langle \texttt{R} \right\rangle_{ctId} \left\langle \texttt{RhoG} \right\rangle_{globalContext} \cdots \right\rangle_{contractInstance}\\ \left\langle \dfrac{\texttt{G:Int}}{\texttt{G +Int G1}} \right\rangle_{gasConsumption}\\ \left\langle \dfrac{\texttt{ListItem(G1) GasList:List}}{\texttt{GasList}} \right\rangle_{gasStack}\end{array}$$

function quantifiers, namely specifiers and modifiers, which may modify the function body and have an impact on the function execution. For instance, modifiers rewrite the function body by replacing "_;" that appears there with the function body. Finally, callFunBody executes the function body that has been modified by function quantifiers.

There are three sub-steps in CALL-FUNCTION-BODY. The first one funBody is the execution of the function body B which is obtained by mapping the cell function with Id N. The second updateReturnParams binds the return parameter with the return value. For instance, if a function returns 1, the value of its return parameter should be 1. The last one updateReturnValue returns the value of the return parameter if there is no return statement in this function.

Statements in a function body are executed sequentially. In the rule FUNCTION-BODY, every time the first statement S in a list of statements is extracted for execution through exeStmt and the remaining statements Ss are processed with this rule recursively until the list of statements becomes empty.

I. ReturnContext

RETURN-CONTEXT is the last step in FUNCTION-CALL to return to the caller instance. In order to finish this function call, the recipient instance R is popped out of contractStack to switch back to the caller instance C. In the meantime, the state information for this function call is removed from functionStack. Furthermore, the "msg" information in Msg is rewritten to the previous one M1 which is also popped out of msgStack. In gasConsumption, the gas consumption in the inner call G is added to that in the outer call G1 which is popped out of gasStack. This rule has three sub-steps, namely clearRecipientContext, clearCallerContext, and propagateException, and ends with the return value of this call. To be specific, clearRecipientContext and clearCallerContext remove the local variable contexts in the instances of the recipient and caller, respectively. For the recipient instance, the variable context is set to be RhoG which is obtained from globalContext and only holds the context of state variables in R. For the caller instance, the variable context is set to be the previous one Rho retrieved from the state information in functionStack.

RULE UPDATE-EXCEPTION-STATE

$$\left\langle \frac{\texttt{updateExceptionState()}}{\cdot} \cdots \right\rangle_{k}$$

$$\left\langle \frac{\texttt{ListItem(\#state(\_,\_,\_,\_,\_))}}{\texttt{ListItem(\#state(\_,\_,\_,\_,true))}} \cdots \right\rangle_{functionStack}$$

RULE REVERT-STATE

$$\left\langle \begin{array}{l} \dfrac{\texttt{revertState()}}{\texttt{revertInContracts(PreCNum,0)} \curvearrowright} \\ \texttt{deleteNewContracts(PreCNum,CNum)} \end{array} \cdots \right\rangle_{k}$$

$$\left\langle \texttt{ListItem(\#state(\_,\_,\_,PreCNum,\_))} \cdots \right\rangle_{functionStack}$$

$$\left\langle \texttt{CNum} \right\rangle_{cntContracts}$$

RULE RETURN-VALUE

$$\left\langle \frac{\texttt{return E:Value}}{\texttt{1}} \cdots \right\rangle_{k}$$

$$\left\langle \frac{\begin{array}{c}\texttt{ListItem(\#state(\_,\_,} \\ \texttt{\#return(\_,\_),\_,\_))}\end{array}}{\begin{array}{c}\texttt{ListItem(\#state(\_,\_,} \\ \texttt{\#return(true,E),\_,\_))}\end{array}} \cdots \right\rangle_{functionStack}$$

RULE RETURN

$$\left\langle \frac{\texttt{return}}{\texttt{1}} \cdots \right\rangle_{k}$$

$$\left\langle \frac{\begin{array}{c}\texttt{ListItem(\#state(\_,\_,} \\ \texttt{\#return(\_,\_),\_,\_))}\end{array}}{\begin{array}{c}\texttt{ListItem(\#state(\_,\_,} \\ \texttt{\#return(true,true),\_,\_))}\end{array}} \cdots \right\rangle_{functionStack}$$

propagateException deals with the propagation of exceptions based on the exception flag recorded in functionStack. If an exception is encountered in a function call, it should be propagated to the call stemming from the "Main" contract which is considered as an independent transaction. In this way, if an exception appears in any nested call, the whole transaction throws and the states of all the involving instances should revert to the ones before this transaction. Due to limit of space, the detailed steps are omitted. Lastly, RETURN-CONTEXT returns the return value of this function call E which is obtained from #return in functionStack.

J. Exception Handling

In UPDATE-EXCEPTION-STATE, the exception flag, the last field in #state is rewritten to true to indicate that an exception has been encountered. In REVERT-STATE, two sub-steps are processed to revert to the state before the transaction. These sub-steps are revertInContracts and deleteNewContracts. To be specific, revertInContracts(PreCNum,0) deals with the reversion to the previous states of the contract instances that were created before this transaction. Particularly, the reversion starts from the instance with Id 0 and ends at the one with Id PreCNum - 1 where PreCNum is the number of instances created before this transaction which is recorded in functionStack. deleteNewContracts(PreCNum,CNum) deletes the new contract instances created in this transaction. Here, CNum is the current number of contract instances retrieved from cntContracts. The reversion to previous states is simply the rewriting of cell contents to previous ones and the deletion of new contract instances is the deletion of a set of cells. Due to limit of space, detailed steps are omitted here.

RULE EXE-STATEMENT

$$\left\langle \frac{\texttt{exeStmt(S:NoBlockStatement)}}{\texttt{S}} \cdots \right\rangle_{k}$$

$$\left\langle \begin{array}{c}\texttt{ListItem(\#state(\_,\_,} \\ \texttt{\#return(false,\_),\_,false))}\end{array} \cdots \right\rangle_{functionStack}$$

RULE EXE-STATEMENT-END

$$\left\langle \frac{\texttt{exeStmt(S:NoBlockStatement)}}{\cdot} \cdots \right\rangle_{k}$$

$$\left\langle \begin{array}{c}\texttt{ListItem(\#state(\_,\_,} \\ \texttt{\#return(ReturnFlag,\_),} \\ \texttt{\_,ExceptionFlag))}\end{array} \cdots \right\rangle_{functionStack}$$

$$\texttt{requires (ReturnFlag ==Bool true)}$$
$$\texttt{orBool (ExceptionFlag ==Bool true)}$$

RULE EXE-STATEMENT-MAIN-CONTRACT

$$\left\langle \frac{\texttt{exeStmt(S:NoBlockStatement)}}{\texttt{S}} \cdots \right\rangle_{k}$$

$$\left\langle \texttt{.List} \right\rangle_{functionStack}$$

K. Return

When return is encountered, the return flag, the first field in #return, is set to be true in functionStack. Particularly, if a value E is returned, the return value, the second field in #return, is rewritten to E. If there is no value to return, we assign true to the return value to indicate that this function call is successful. Both RETURN-VALUE and RETURN end with the integer 1 which indicates the end of the expression. Any integer followed by ";" will be rewritten to ·, which represents the end of the return statement.

L. Statements

As shown in EXE-STATEMENT, the execution of each statement in the function body is affected by the return and exception flags in functionStack. Generally speaking, a statement will be executed when both of the two flags are false, indicating that neither return nor an exception has been encountered. For statements in the "Main" contract, we simply execute them.

Please note that we limit the statement for execution to NoBlockStatement where no block structures are present. This is because we need to exclude block structures to keep all the other statements in the function body parallel to return and exception statements. In this way, each statement can be executed sequentially without any nested structures. Once return or an exception is encountered, the execution stops regardless of the structure of the statements. Statements with blocks, named as BlockStatements, can be transformed into a list of NoBlockStatements. Due to limit of space, the transformation rules are omitted.