Precise Dynamic Symbolic Execution for Nonuniform Data Access in Smart Contracts

Jianjun Huang, Jiasheng Jiang, Wei You, and Bin Liang

Abstract—Dynamic symbolic execution (DSE) has been successfully adopted for vulnerability detection in desktop and mobile platforms. Unfortunately, we cannot simply extrapolate those techniques to smart contracts. The major challenge is that smart contracts exhibit a nonuniform data access mode. Other than accessing the data via uniform addresses, smart contracts compromise multiple addressing modes, including flat address mode and key-value mode. More seriously, accessing a key-value table usually involves additional hash operations to obtain the keys. In this paper, we propose a DSE framework to resolve the nonuniform data access in smart contracts. More specifically, we exactly track the symbolic variables with concrete addresses and compute the actual/hash keys for table-like accesses. We also take the symbolic keys into account to distinguish data accesses incidentally with the same concrete keys resulting from artificially generated values. We describe the DSE framework in operational semantics. On top of the framework, we implement an integer overflow detector NOVA and a multi-transactional vulnerability detector MTVD. The experiments show that NOVA outperforms state-of-the-art analysis tools in detecting the integer overflows with much higher precision and recall, 94.2% and 93.0%, respectively. MTVD successfully reports three ether leaking vulnerabilities and one suicidal issue from real-world smart contracts.

Index Terms—Smart Contracts, Nonuniform Data Access, Dynamic Symbolic Execution, Operational Semantics, Integer Overflow

1 INTRODUCTION

AFTER the introduction of Bitcoin, the cryptocurrencies have attracted a huge number of people. The second largest blockchain platform, Ethereum, supports to run smart contracts for distributing and investing the tokens, a special type of virtual coins. The Ethereum smart contracts are mainly developed in Solidity and compiled into bytecode to run in the Ethereum virtual machine (EVM). Our investigation shows that, more than two million of smart contracts have been deployed in Ethereum so far.

Despite the success, the vulnerabilities in smart contracts have resulted in quite a lot of damages. For example, a reentrancy bug in the DAO contract led to a loss of more than 60 million of dollars [1] and the exploitation of an integer overflow in the contract of BeautyChain Coin (BEC) made all the tokens worthless [2].

It has become emergent to detect the vulnerabilities in smart contracts. Fortunately, dynamic symbolic execution (DSE) has achieved great success for vulnerability detection in desktop and mobile platforms [3], [4], [5], [6]. However, directly extrapolating those mature techniques to smart contracts is nontrivial. Different with traditional platforms on which the data are uniformly accessed with the same addressing mechanism, smart contracts compromise multiple addressing modes. We call it the *nonuniform data access* mode. More specifically, while some data are stored in flat addressed regions and accessible via general addresses, some other data are placed in a key-value table [7]. Retrieving a table slot requires a valid key and in many cases, the key is obtained through hash operations on several data fields. The obvious difference impedes our attempt of simply employing the DSE solution from other platforms and inspires us to propose a DSE engine that can well resolve the nonuniform data access nature so as to precisely track the data flows symbolically.

In this paper, we present a precise and comprehensive DSE framework targeting smart contracts and formalize the approach in operational semantics. The framework addresses the nonuniform data access problem in three folds. First, with artificially generated values, a concrete execution operates the data in flat addressed regions with certainty and thus the framework can track the symbolic variables accurately, eliminating any nondeterminacy. Second, concrete values and their hash results are used to position the slots in the key-value table. By this means, we can easily discover relevant slots of symbolic expressions and multiple accesses to the same slot can be correctly connected. Third, we take the potentially symbolic representations as a second level key for accessing a table slot, to address the challenge of positioning the same slot for different queries when the artificial values are the same.

On top of the DSE framework, we implement an integer overflow detector, NOVA, and apply it to 21,016 smart contracts that have been deployed in Ethereum. We verify 200 open-sourced smart contracts in Remix [8], the IDE for smart contract development and testing, and the results show that NOVA achieves high precision and recall, 94.2% and 93.0%, respectively. We also compare NOVA with four state-of-the-art tools, OYENTE [9], MYTHRIL [10], OSIRIS [11] and VERISMART [12]. The comparisons demonstrate that NOVA drastically reduces the false positives while detecting much more real vulnerabilities. Furthermore, we implement

• *J. Huang, J. Jiang, W. You and B. Liang are with the School of Information, Renmin University of China, Beijing 100872, China; and also with Key Laboratory of DEKE (Renmin University of China), MOE, China. E-mail: {hjj, jjscool, youwei, liangb}@ruc.edu.cn.*

Manuscript received XXXX, 2020; revised XXXX, 2021; accepted XXXX 2021. Date of publication XXXX 2021; date of current version XXXX 2021. (Corresponding author: Bin Liang.)
Recommended for acceptance by
Digital Object Identifier no.

MTVD to detect two kinds of multi-transactional vulnerabilities (i.e., ether leaking and suicidal), and MTVD reports four vulnerabilities from the collected open-source contracts.

Our work makes the following major contributions.

- We propose a dynamic symbolic execution framework for smart contracts. To the best of our knowledge, our work is the first to present a DSE framework for precisely resolving the nonuniform data accesses in smart contracts.
- We present operational semantics to formalize the working mechanism of the framework. The proposed semantics provide rich information for DSE and handling the nonuniform data accesses.
- We implement an integer overflow detector NOVA and evaluate it on real-world smart contracts. We verify 200 smart contracts with the exploits generated by NOVA in Remix [8], compare NOVA with state-of-the-art tools and demonstrate its effectiveness with low false positive and false negative rates.
- We implement MTVD to detect multi-transactional vulnerabilities. The experimental results show that our DSE framework is capable to be extended for more-challenging vulnerability detection.

2 Background and Motivation

We use the simplified smart contract in Figure 1 to talk about the background and motivate our technique.

Smart contract. Ethereum smart contracts are usually developed in Solidity, as shown in Figure 1(a). The contract `MVToken` contains a field `balances` and a function `transfer`. The former records a *mapping* from the users' account to the number of tokens they hold and the latter transfers the tokens from a sender to a list of recipients, each for the amount of `value` and charged certain amount of `fee`. The developers introduce a `SafeMath` library to ensure safe arithmetic operations. Line 2 presents an example of safe `addition`. An `assert` is placed after the addition, which can break down the execution and roll back all changes if the condition is unsatisfactory, guaranteeing no integer overflows for the addition in a normal execution. The safe addition is invoked at line 13. Line 15 examines the conditions for subsequent operations. Violating the conditions results in abnormal termination and state reverting, as done by the `assert`.

EVM and nonuniform data access. The smart contract is compiled into bytecode and then executed in EVM. As shown in Figure 1(b), during an execution, EVM maintains four data regions, the read-only *input* region, a runtime *stack*, a transient *memory* and a persistent *storage*. The first three regions are alive only when a smart contract is being executed while the storage holds the global state of the contract, which spans the life across different executions.

While in desktop programs, all memory accesses are carried out through uniform addresses, the data regions in EVM show a nonuniform access mode. The *stack* behaves as what its name shows. The *input* and the *memory* are byte arrays starting at address zero. The input contains a four-byte signature that directs the execution to the specific function and a series of 32-byte data, following the order of the formal parameters. If there exist variable-length arrays, e.g., `to`, the corresponding slot saves the offset to the actual array content, the length of the array and the elements. The memory is used to temporarily store data chunks which usually exceed 256 bytes, the data size of a stack element. The arrays in the input are copied to the memory and then loaded to the stack for operations. The *storage* is organized as a key-value table [7]. Simple fields can be directly accessed with their compiler-determined indices. Complex fields like the mapping `balances` require hash values as the keys. For example, to retrieve the data for `balances[from]`, EVM computes a hash value of $from \cdot index_{balances}$, where the symbol '$\cdot$' denotes a concatenation, and visits the slot corresponding to the hash value.

Symbolic execution. Static symbolic executions have been widely used to capture the bugs in smart contracts [9], [10], [13]. However, Frank et al. have shown in a recent study that state-of-the-art tools generally encounter problems for precisely modeling the nonuniform data accesses [14]. For instance, an array in the input involving symbolic offset and length can make the memcopy operation nondeterministic. An example is the `multiTransfer(address[], uint[])` function in RocketCoin [15]. Symbolically copying the second array from input to memory encounters a symbolic offset which is determined by the symbolic length of the first array. In addition, storage accesses usually involve hash operations and ineffective handling would obviously influence the analysis.

Dynamic symbolic execution (DSE) can help address the above issues with concrete execution which provides actual values for memcopy and hash operations. Symbolic execution engine can utilize these concrete values to precisely label and differentiate the symbolic variables. However, the nonuniform data accesses prevent us from employing a mature DSE technique for traditional programs and motivate us to develop a smart contract targeted DSE technique.

Integer overflow. Now we pay attention to one of the most dangerous vulnerabilities in smart contracts, the integer overflow vulnerability. In a recent report, integer underflow and overflow are ranked as two of the four biggest vulnerabilities in smart contracts [16]. Specifically, the integer underflow/overflow (*overflow* for brevity in this paper) vulnerabilities account for 95.3% of the high-severity instances found by the analysts. Therefore, in this paper, we mainly focus on the integer overflow vulnerability.

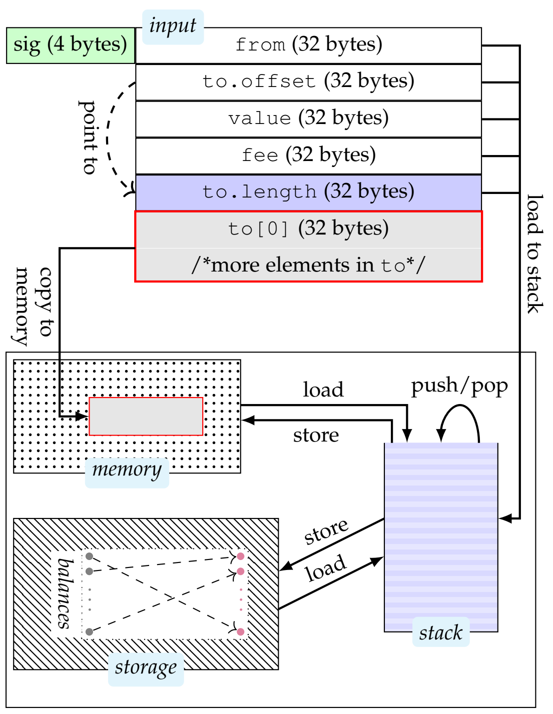
The smart contract in Figure 1(a) contains four potentially vulnerable arithmetic operations, two additions at lines 3 and 19, a subtraction at line 16 and a multiplication at line 14. Carefully auditing the code, we can exclude the addition within the safe math library and the subtraction, because they are protected by a post assertion or a pre-requirement. The developers may forget to protect the other two operations, leaving a chance of being exploited via the integer overflow vulnerabilities. The BeautyChain Token is a famous example [2].

In our example, if an adversary executes the function with two recipients, `value` $= 2^{255}$ and `fee` $= 1$, `amount` at line 14 will be just 2 because the multiplication result exceeds the maximum integer $2^{256}-1$. Such a small `amount` easily passes the requirement check at line 15. The sender pays only two tokens (line 16) while each recipient can

```
1  library SafeMath {
2     function add(uint a, uint b) internal returns (uint) {
3        uint c = a + b;
4        assert(c >= a);
5        return c;
6     }
7  }
8  contract MVToken {
9     using SafeMath for uint;
10    mapping (address => uint) balances;
11    function transfer(address from, address[] to, uint
           value, uint fee) public returns (bool) {
12       uint count = to.length;
13       uint each = value.add(fee);
14       uint amount = count * each;
15       require(amount > 0 && balances[from] >= amount);
16       balances[from] = balances[from] - amount;
17       for (uint i = 0; i < count; i++) {
18          address receiver = to[i];
19          balances[receiver] = balances[receiver] + value;
20       }
21       // other operations are omitted
22    }
23 }
```

call the function

(a) Example smart contract code.



(b) Nonuniform data accesses for a running contract.

Fig. 1. Motivating example. On the left is a simplified smart contract and on the right is the nonuniform data accesses when the contract is running.

obtain 2^{255} tokens. Because too many tokens emerge all of a sudden but the total value of the tokens are not changed accordingly, all other investors will find their holdings are worth nothing.

Though Ghelab et al. have demonstrated that state-of-the-art bug-finding tools can emit high false positives and false negatives [17], we detect the vulnerabilities in the example with two bug-finders, OYENTE [9] and MYTHRIL [10], and manually inspect the results. OYENTE reports no real problems but one false alarm with two missed vulnerabilities and MYTHRIL correctly uncovers the suspicious multiplication overflow but misses the addition at line 19. Besides, MYTHRIL reports an overflow involving the array size calculation, but in practice, the arrays inside the input are generally small and their sizes in bytes are far less than the maximum integer in smart contract.

Our solution. We build a DSE framework, combining concrete and symbolic executions, which utilizes concrete values to handle nonuniform data accesses. Based on the DSE engine, we implement a precise integer overflow detector NOVA. NOVA leverages the SMT solver Z3 [18] to determine whether certain state (the input or storage values) can pass the safety checks but lead to overflows. For the example contract, NOVA reports two integer overflows without false warnings.

3 Dynamic Symbolic Execution Framework

We propose a DSE framework for smart contracts. More specifically, we describe the operational semantics of the bytecode instructions, which are suitable for dynamic symbolic executions on the EVM bytecode. In general, the DSE framework launches the given bytecode for execution, generates concrete and symbolic values for the inputs and unknown storage fields, and collects the path condition along the execution. At the end of the execution, Z3 is used to search a new feasible path. With the resolved concrete values, the DSE framework restarts another execution and places the values on request so that the execution is directed to the specific path. Repeating the procedure until all feasible paths have been explored, the DSE framework finishes the analysis for the target contract.

In below sections, we first introduce some basic notations and then focus on describing the semantics for path exploration and nonuniform data access handling.

program		::=	*instruction**
instruction	i	::=	POP | PUSH v | ADD | MUL | SUB | LT
			| JUMPI | CALLDATALOAD | CALLDATACOPY
			| MLOAD | MSTORE | SLOAD | SSTORE | SHA3
			| CALL | RETURN | REVERT | SUICIDE | ···
value	v	::=	256-bit unsigned integer

Fig. 2. The language definition for the EVM bytecode.

3.1 Operational Semantics: Basics

Figure 2 defines the language of the contract bytecode. Most instructions do not have their operands explicitly shown in the bytecode. The only exception is PUSH which pushes the given 256-bit integer into the stack. Due to the space limit, we omit many instructions that will not be discussed in this paper.

To ease the discussion, we define symbols for different purposes and use mathematic symbols like '+' and '<' directly in symbolic operations for simplicity. Figure 3 shows

$$\text{LT-T}\ \frac{[(a,b),(p,q),\mathcal{S}'] = \mathcal{S}.pop(2) \quad a < b \quad \mathcal{S}'' = \mathcal{S}'.push(1) \quad s = p < q \quad \mathcal{P}' = is\_symbolic(s)?\mathcal{P}.push(s) : \mathcal{P} \quad \zeta = \Sigma(pc+1)}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{LT} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}'', \mathcal{M}, \mathcal{T}, \mathcal{P}', \mathcal{E}, pc+1, \zeta}$$

$$\text{LT-F}\ \frac{[(a,b),(p,q),\mathcal{S}'] = \mathcal{S}.pop(2) \quad a \geq b \quad \mathcal{S}'' = \mathcal{S}'.push(0) \quad s = p \geq q \quad \mathcal{P}' = is\_symbolic(s)?\mathcal{P}.push(s) : \mathcal{P} \quad \zeta = \Sigma(pc+1)}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{LT} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}'', \mathcal{M}, \mathcal{T}, \mathcal{P}', \mathcal{E}, pc+1, \zeta}$$

$$\text{JUMPI-T}\ \frac{[(d,1),\_,S'] = S.pop(2) \quad \zeta = \Sigma(d)}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{JUMPI} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}', \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, d, \zeta}$$

$$\text{JUMPI-F}\ \frac{[(d,0),\_,S'] = S.pop(2) \quad \zeta = \Sigma(pc+1)}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{JUMPI} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}', \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc+1, \zeta}$$

Fig. 3. Rules for collecting path constraints (unsigned less-than comparison `LT`) and directing the concrete execution (conditional jump `JUMPI`).

the rules for two representative instructions, the unsigned less-than comparison `LT` and the conditional jump `JUMPI`.

When an instruction is being executed, the DSE framework maintains a state of the running smart contracts, consisting of a tuple of the following symbols. Σ maps the indices of all instructions to the corresponding instructions. $\mathcal{C}$, $\mathcal{S}$, $\mathcal{M}$ and $\mathcal{T}$ represent the states for the input, the stack, the memory and the storage, respectively. Each is made up of the concrete state and the symbolic state, distinguished by the subscripts. For example, $\mathcal{C}_c$ records the concrete values in the input and $\mathcal{C}_s$ maps the offsets to the symbolic values. Note that *symbolic* containers, e.g., $\mathcal{S}_c$, can hold concrete values as well. We use $\mathcal{P}$ to hold the path condition that current execution has passed and maintain in $\mathcal{E}$ the heuristically inferred constraints for practical enhancement, e.g., a restriction of the array length. The index of current instruction is denoted by pc. With the concrete and symbolic states, the framework performs the operations and validates the checks above the line in the rule.

We collect a path constraint when executing `LT`. Following the semantics, we pop two elements from the stack and evaluate the comparison. In the notation, (a, b) denotes the pair of concrete values, p and q represent the elements from the symbolic stack and $\mathcal{S}'$ maintains the remaining elements. If $a < b$ holds in LT-T, we generate a *symbolic* expression, save it to $\mathcal{P}$ and fetch next instruction for execution. Note that, instead of saving the expression "$p < q$" to $\mathcal{S}_s$, we use the concrete value *one* as the expression should always evaluate to be true and could probably slow down the constraint solving when used in other expressions.

The instruction `JUMPI` pops two elements from the stack. If the second element is equal to one, the execution moves to the instruction at index d (JUMPI-T). Otherwise, the sequentially next instruction is to be executed (JUMPI-F). We use the symbol '_' to indicate that the symbolic elements are uninteresting for this instruction.

3.2 Path Exploration and Execution Restart

As we mentioned earlier, the DSE framework explores a new feasible path at the end of an execution and then restarts another execution targeting the specific path. In smart contracts, a path can be terminated normally or abnormally. A normal termination keeps all changes to the storage in the blockchain and an abnormal termination discards any changes and reverts the state. `RETURN` and `REVERT` are typical instructions for each case separately. We display only the semantics for `RETURN` in Figure 4, but it is straightforward to replace `RETURN` with the other path-terminating instructions to deduce corresponding rules.

We explore new paths by manipulating the gathered path condition $\mathcal{P}$. An empty $\mathcal{P}$ indicates no more paths (RETURN-N) and the framework finishes the analysis with an invalid index (-1) and a nonexistent instruction ($\perp$). If any path constraint exists, we perform a depth first search algorithm, negating the latest path constraint and validating if the modified path condition represents a new path. If not, we check the remaining path constraints (RETURN-E). In other words, the RETURN-family rules are applied to the same instruction with the updated path condition. RETURN-F handles the case of seeing a new but infeasible path and RETURN-T restarts an execution for a new feasible path by resetting next instruction. It is notable that, we restrict the constraint solving with the heuristically obtained constraints $\mathcal{E}$ so that we will never start an execution with impractical concrete values, e.g., an array length of 2^{255}. The concrete values resolved by Z3 are taken along with the restarted execution.

3.3 Input Loading

The instruction `CALLDATALOAD` loads a 32-byte chunk from the input to the stack for later operations. Before we dive into the details, it is noted that the compiler will not generate bytecode that reads data from the input region at the same offset more than once. Namely, each `CALLDATALOAD` processes a distinctive part in the input. We have four rules to model the instruction, as shown in Figure 5.

In most cases, the DSE framework executes `CALLDATALOAD` with completely concrete offset in the input, e.g., loading `from` and `to.offset` in Figure 1. The rule CDL-C-N is applied when no concrete value has been assigned to the specific concrete offset. We use $\mathcal{C}_c \vdash a :\perp$ to indicate that in $\mathcal{C}_c$, a maps to nothing. We artificially generate concrete data and associate a new symbolic variable to the offset. The notation $[a : c]\mathcal{C}_c$ means to update $\mathcal{C}_c$ with a new mapping relation $a \rightarrow c$.

When arrays exist, e.g., `to` in Figure 1, the concrete values are not enough in the first execution. Because we know nothing about the input structure, a generated array offset can be invalid and thus lead to incorrect access to the array content. We handle this case in CDL-S-N with heuristics. If we see a symbolic offset, we infer that we encounter an array access and the target denotes the length of the array. The corresponding concrete and symbolic values are created and saved. We do not update $\mathcal{C}$ because the provided concrete offset a_c is potentially invalid. The actual array offset is heuristically computed at the end of an execution (see Figure 4), based on the structure of the input and the

$$\text{RETURN-N}\;\frac{is\_empty(\mathcal{P})}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{RETURN} \Rightarrow \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, -1, \bot}$$

$$\text{RETURN-E}\;\frac{not\_empty(\mathcal{P}) \quad [s, \mathcal{P}'] = \mathcal{P}.pop() \quad S = \mathcal{P}' \wedge \neg s \quad \neg is\_new\_path(S)}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{RETURN} \Rightarrow (\texttt{RETURN-N/E/F/T}\ only)\ \Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}', \mathcal{E}, pc, \texttt{RETURN}}$$

$$\text{RETURN-F}\;\frac{not\_empty(\mathcal{P}) \quad [s, \mathcal{P}'] = \mathcal{P}.pop() \quad S = \mathcal{P}' \wedge \neg s \quad is\_new\_path(S) \quad solve(S \wedge \mathcal{E}) = \emptyset}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{RETURN} \Rightarrow (\texttt{RETURN-N/E/F/T}\ only)\ \Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}', \mathcal{E}, pc, \texttt{RETURN}}$$

$$\text{RETURN-T}\;\frac{not\_empty(\mathcal{P}) \quad [s, \mathcal{P}'] = \mathcal{P}.pop() \quad S = \mathcal{P}' \wedge \neg s \quad is\_new\_path(S) \quad [\mathcal{C}'_c, \mathcal{T}'_c] = solve(S \wedge \mathcal{E}) \quad \mathcal{C}'_c \neq \emptyset \quad \zeta = \Sigma(0)}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{RETURN} \Rightarrow \Sigma, \mathcal{C}', \emptyset, \emptyset, \mathcal{T}', \emptyset, \emptyset, 0, \zeta}$$

Fig. 4. Semantics for path exploration and execution restart (`RETURN`).

$$\text{CDL-C-N}\;\frac{\begin{array}{c}[a, a, \mathcal{S}'] = \mathcal{S}.pop() \quad \mathcal{C}_c \vdash a : \bot \quad c = mk\_concrete() \quad s = mk\_symbolic() \quad \mathcal{C}'_c = [a : c]\mathcal{C}_c \quad \mathcal{C}'_s = [a : s]\mathcal{C}_s \\ \mathcal{S}''_c = \mathcal{S}'_c.push(c) \qquad \mathcal{S}''_s = \mathcal{S}'_s.push(s) \qquad \zeta = \Sigma(pc + 1)\end{array}}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{CALLDATALOAD} \Rightarrow \Sigma, \mathcal{C}', \mathcal{S}'', \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc + 1, \zeta}$$

$$\text{CDL-S-N}\;\frac{\begin{array}{c}[a_c, a_s, \mathcal{S}'] = \mathcal{S}.pop() \quad is\_symbolic(a_s) \quad c = mk\_concrete() \quad \mathcal{S}''_c = \mathcal{S}'_c.push(c) \quad s = mk\_symbolic() \quad \mathcal{S}''_s = \mathcal{S}'_s.push(s) \\ a_o = back\_for\_offset(a_s) \qquad \mathcal{E}' = \mathcal{E}.push((a_o < \mathbb{X}) \wedge (s < \mathbb{Y})) \qquad mark\_array\_info(a_o, a_s, s) \qquad \zeta = \Sigma(pc + 1)\end{array}}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{CALLDATALOAD} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}'', \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}', pc + 1, \zeta}$$

$$\text{CDL-R-N}\;\frac{\begin{array}{c}[a, \_, \mathcal{S}'] = \mathcal{S}.pop() \qquad \mathcal{C}_c \vdash a : c \qquad \mathcal{C}_s \vdash a : \bot \qquad \neg is\_location\_of\_array\_offset(a) \\ s = mk\_symbolic() \quad \mathcal{S}''_c = \mathcal{S}'_c.push(c) \quad \mathcal{S}''_s = \mathcal{S}'_s.push(s) \quad \mathcal{C}'_s = [a : s]\mathcal{C}_s \quad \zeta = \Sigma(pc + 1)\end{array}}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{CALLDATALOAD} \Rightarrow \Sigma, \mathcal{C}', \mathcal{S}'', \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc + 1, \zeta}$$

$$\text{CDL-R-A}\;\frac{\begin{array}{c}[a, \_, \mathcal{S}'] = \mathcal{S}.pop() \quad \mathcal{C}_c \vdash a : c \quad \mathcal{C}_s \vdash a : \bot \quad is\_location\_of\_array\_offset(a) \\ \mathcal{S}'' = \mathcal{S}'.push(c) \qquad \mathcal{C}'_s = [a : c]\mathcal{C}_s \qquad \zeta = \Sigma(pc + 1)\end{array}}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{CALLDATALOAD} \Rightarrow \Sigma, \mathcal{C}', \mathcal{S}'', \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc + 1, \zeta}$$

Fig. 5. Rules for `CALLDATALOAD`, which moves the data from the input region to the stack.

number of input accesses. We perform a backward search on the symbolic offset a_s and acquire the symbolic variable a_o that is associated with the offset value in the input. For an array in the input, we should see $a_s = a_o + 4$. We restrict the array offset and length for practical consideration and insert corresponding constraints into $\mathcal{E}$. In addition, we mark the array related representations such that we can build a correct input with the arrays.

In a restarted execution in which proper concrete data have been constructed by RETURN-T for an offset but the data do not indicate the offset of an array, we apply CDL-R-N to update the states. If the array offset is encountered, e.g., `to.offset` in Figure 1, instead of generating a symbolic variable, we use the actual offset afterwards (CDL-R-A) to relax any correlated symbolic operations.

3.4 Memory Operations

Memory operations essentially involve three instructions. `CALLDATACOPY` copies data from the input to the memory. `MLOAD` transfers data from the memory to the stack and `MSTORE` reverses the data movement. Figure 6 presents the operational semantics for the three instructions.

The rule CDC-S is related to the condition in which the array offset has not been concretely determined. In other words, the corresponding `CALLDATACOPY` is executed after a dependent `CALLDATALOAD` that was applied CDL-S-N. We fake an array according to the provided size and record the values in the memory. When we are processing a known array after CDL-R-A, we apply the rule CDC-C to generate symbolic variables for the elements and any unknown concrete values and update the memory. For brevity, we use d_i to denote the input offset $d + i * 32$ and let $m_i = m + i * 32$.

The next three rules in Figure 6 model the data flows between the memory and the stack, corresponding to `MLOAD` and `MSTORE`. MLOAD-C processes fully concrete data without any symbolic value associated and MLOAD-S retrieves both the concrete and symbolic values, e.g., an array element copied from the input. MSTORE simply overwrites the memory records with the given values.

The CDL-* (in Figure 5) and CDC-* rules together help the DSE framework precisely model any arrays in the input and track the corresponding memcopy operations accurately. The problem of nondeterministic array read/write faced by lightweight static symbolic executions can be easily overcome in our DSE framework. Besides, concrete offsets make it more efficient to decide if multiple operations via `MLOAD`/`MSTORE` indeed process the same chunk, compared to the fully symbolic execution approaches [14].

3.5 Storage Accesses

The storage is organized as a key-value table. `SLOAD` and `SSTORE` accesses the corresponding slot in the table with a given key. Simple fields are associated to constant keys that are determined at the compilation time. Complex fields pass a hash operation to generate the keys. Concrete execution can do the hash operation and obtain a real hash value. However, it is still challenging to precisely track the symbolic expressions and update the concrete storage region if we simply adopt the original key-value table representation.

To cope with the problem, we design a two-fold access model for the storage, taking the involved expression as the second level indicator of a data slot and the concrete key used by `SLOAD`/`SSTORE` as the first level indicator. Take Figure 7 as an instance. The DSE framework will maintain the storage in a form of $(X \longrightarrow \{\{Y \rightarrow bal_{from}\}, \{Z \rightarrow bal_{to}\}\})$ at the end of the function, where bal_x denotes the symbolic expression of `balances[x]`. X is a concrete

$$\text{CDC-S}\;\frac{\begin{array}{c}[(m,d_c,l),(\_,d_s,\_),\mathcal{S}']=\mathcal{S}.pop(3)\quad is\_symbolic(d_s)\quad n=\lceil\frac{l}{32}\rceil\quad 0\le i<n\quad [..,c_i,..]=mk\_concrete(n)\\ [..,s_i,..]=mk\_symbolic(n)\quad mark\_array\_elements([...,s_i,...])\quad \mathcal{M}_c'=[m_i:c_i]\mathcal{M}_c\quad \mathcal{M}_s'=[m_i:s_i]\mathcal{M}_s\quad \zeta=\Sigma(pc+1)\end{array}}{\Sigma,\mathcal{C},\mathcal{S},\mathcal{M},\mathcal{T},\mathcal{P},\mathcal{E},pc,\texttt{CALLDATACOPY}\Rightarrow\Sigma,\mathcal{C},\mathcal{S}',\mathcal{M}',\mathcal{T},\mathcal{P},\mathcal{E},pc+1,\zeta}$$

$$\text{CDC-C}\;\frac{\begin{array}{c}[(m,d,l),(\_,d,\_),\mathcal{S}']=\mathcal{S}.pop(3)\quad n=\lceil\frac{l}{32}\rceil\quad 0\le i,j,x<n\quad \mathcal{C}_c\vdash d_j:c_j\quad \forall(x\neq j),c_x=mk\_concrete()\\ [..,s_i,..]=mk\_symbolic(n)\quad mark\_array\_elements([...,s_i,...])\quad \mathcal{M}_c'=[m_i:c_i]\mathcal{M}_c\quad \mathcal{M}_s'=[m_i:s_i]\mathcal{M}_s\quad \zeta=\Sigma(pc+1)\end{array}}{\Sigma,\mathcal{C},\mathcal{S},\mathcal{M},\mathcal{T},\mathcal{P},\mathcal{E},pc,\texttt{CALLDATACOPY}\Rightarrow\Sigma,\mathcal{C},\mathcal{S}',\mathcal{M}',\mathcal{T},\mathcal{P},\mathcal{E},pc+1,\zeta}$$

$$\text{MLOAD-C}\;\frac{[a,\_,\mathcal{S}']=\mathcal{S}.pop()\quad \mathcal{M}_s\vdash a:\perp\quad \mathcal{M}_c\vdash a:c\quad \mathcal{S}''=\mathcal{S}'.push(c)\quad \zeta=\Sigma(pc+1)}{\Sigma,\mathcal{C},\mathcal{S},\mathcal{M},\mathcal{T},\mathcal{P},\mathcal{E},pc,\texttt{MLOAD}\Rightarrow\Sigma,\mathcal{C},\mathcal{S}'',\mathcal{M},\mathcal{T},\mathcal{P},\mathcal{E},pc+1,\zeta}$$

$$\text{MLOAD-S}\;\frac{[a,\_,\mathcal{S}']=\mathcal{S}.pop()\quad \mathcal{M}_s\vdash a:s\quad \mathcal{M}_c\vdash a:c\quad \mathcal{S}_c''=\mathcal{S}_c'.push(c)\quad \mathcal{S}_s''=\mathcal{S}_s'.push(s)\quad \zeta=\Sigma(pc+1)}{\Sigma,\mathcal{C},\mathcal{S},\mathcal{M},\mathcal{T},\mathcal{P},\mathcal{E},pc,\texttt{MLOAD}\Rightarrow\Sigma,\mathcal{C},\mathcal{S}'',\mathcal{M},\mathcal{T},\mathcal{P},\mathcal{E},pc+1,\zeta}$$

$$\text{MSTORE}\;\frac{[(m,c),(\_,s),\mathcal{S}']=\mathcal{S}.pop(2)\quad \mathcal{M}_c'=[m:c]\mathcal{M}_c\quad \mathcal{M}_s'=[m:s]\mathcal{M}_s\quad \zeta=\Sigma(pc+1)}{\Sigma,\mathcal{C},\mathcal{S},\mathcal{M},\mathcal{T},\mathcal{P},\mathcal{E},pc,\texttt{MSTORE}\Rightarrow\Sigma,\mathcal{C},\mathcal{S}',\mathcal{M}',\mathcal{T},\mathcal{P},\mathcal{E},pc+1,\zeta}$$

Fig. 6. Semantic rules for the memory operations, including memcopy (`CALLDATACOPY`), read (`MLOAD`) and write (`MSTORE`).

```
1 function transferFrom(address from, address to,
      uint value) ... {
2     require(value > 0 && value <= balances[from]);
3     balances[from] = balances[from] - value;
4     balances[to] = balances[to] + value;
5 }
```

Fig. 7. A vulnerable function in an ERC20 contract.

value, with which `SLOAD` and `SSTORE` visit the storage slots, e.g., the result of $sha3(0,0)$ in the above example. Y and Z are the corresponding symbolic expressions, i.e., $from$ and *to*, respectively.

We further refer to Figure 7 and briefly explain how an symbolic execution works with naive key-value mappings to concretely illustrating the necessity of the two-fold access model. Our DSE engine initially generates a concrete value 0x0 for every input field. Namely, `from` and `to` are assigned 0x0. Symbolic executions will not update their values since they are not involved in any conditional checks. When the DSE is about to execute line 3, the path constraint corresponding to the earlier `require` is $(value > 0) \wedge (value \leq bal_{from})$, in which we use $value$ to represent the symbolic input of the parameter `value` and bal_{from} to denote the symbolic variable for `balances[from]`, for simplicity. According to smart contract semantics, the DSE framework computes a hash value via an `SHA3` instruction on the concatenation of the index of `balances` (e.g., 0x0) and the value of `from` (0x0), i.e., $sha3(0,0)$. Later, the DSE engine looks up the corresponding value of the concrete hash value in the storage via an `SLOAD` instruction. Nothing is found and thus a symbolic variable *bf* is created and bound to the corresponding storage slot. The execution of line 3 will update the slot with a new symbolic expression $(bal_{from} - value)$ upon an `SSTORE` instruction. To fetch `balance[to]`, a hash operation as $sha3(0,0)$ is performed as `to` is assigned 0x0 as well. Though `balances[from]` and `balances[to]` should generally refer to different storage slots, the DSE framework here treats them the same, provided the same concrete hash. Hence, the previously stored expression $(bal_{from} - value)$ is retrieved for `balances[to]` and the addition is symbolically emulated as $((bal_{from} - value) + value)$. Such a symbolic execution schema will obviously lead to a missed vulnerability at line 4 in Figure 7 if an integer overflow checker is built on the framework.

With the two-fold model, however, we can distinguish bal_{from} from bal_{to} based on the second level key, i.e., the symbolic $from$ and to, even if both are assigned 0x0 in the concrete execution. We propose the semantic rules in Figure 8, handling various cases.

For the first time of looking up a value in the storage, we apply SLOAD-N, creating a new symbolic representation and updating the storage. If a concrete key is revisited, the framework compares the given symbolic expression with each key (i.e., Y and Z in the above example) in the second level mappings. A fresh a_s denotes a new slot access and we generate values for it (SLOAD-S-RN). Otherwise, we reuse existing values (SLOAD-S-RR). The functions *sat* and *unsat* utilize the SMT solver to test whether the given expression is satisfactory or not. Though not detailed in the rule SSTORE, the framework performs the same checking when updating a storage field.

SLOAD-C-RR is applied when a fully concrete key is reused for storage access in current execution. In a restarted execution where the target field may have been assigned a concrete value at the first visit, we generate only the corresponding symbolic variable (SLOAD-C-RN). V_s in SLOAD-S-* rules can also be $\perp$ but we omit the cases due to the space limit.

The next two rules handle the hash instruction `SHA3`. Completely concrete data are directly hashed, as displayed in SHA3-C. If any part of the data is symbolic, we concatenate the concrete and symbolic parts to make the symbolic key used for a later storage access.

Recall the vulnerable function in Figure 7. With the two-fold storage access model, our DSE framework symbolically performs the addition at line 4 as $(bal_{to} - value)$ constrained by $(value > 0) \wedge (value \leq bal_{from})$. As bal_{to} and bal_{from} are two arbitrary symbolic variables, an integer overflow checker on top of the DSE framework can easily uncover the potential addition overflow vulnerability.

3.6 Multi-transactional Execution

The framework can also support multi-transactional executions. We show the rule MT in Figure 9. If an execution reaches the end of current path (e.g., `RETURN` and `REVERT`),

$$\text{SLOAD-N}\;\frac{\begin{array}{c}[a_c, a_s, \mathcal{S}'] = \mathcal{S}.pop() \quad \mathcal{T}_c \vdash a_c :\perp \quad c = mk\_concrete() \quad s = mk\_symbolic() \quad \mathcal{S}''_c = \mathcal{S}'_c.push(c) \\ \mathcal{S}''_s = \mathcal{S}'_s.push(s) \quad \mathcal{T}'_c = [a_c : \{a_s \to c\}]\mathcal{T}_c \quad \mathcal{T}'_s = [a_c : \{a_s \to s\}]\mathcal{T}_s \quad \zeta = \Sigma(pc+1)\end{array}}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{SLOAD} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}'', \mathcal{M}, \mathcal{T}', \mathcal{P}, \mathcal{E}, pc+1, \zeta}$$

$$\text{SSTORE}\;\frac{\begin{array}{c}[(a_c, v_c), (a_s, v_s), \mathcal{S}'] = \mathcal{S}.pop(2) \quad \mathcal{T}_c \vdash a_c : V_c \quad V'_c = [a_s : v_c]V_c \quad \mathcal{T}'_c = [a_c : V'_c]\mathcal{T}_c \\ \mathcal{T}_s \vdash a_c : V_s \quad V'_s = [a_s : v_s]V_s \quad \mathcal{T}'_s = [a_c : V'_s]\mathcal{T}_s \quad \zeta = \Sigma(pc+1)\end{array}}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{SSTORE} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}', \mathcal{M}, \mathcal{T}', \mathcal{P}, \mathcal{E}, pc+1, \zeta}$$

$$\text{SLOAD-C-RR}\;\frac{\begin{array}{c}[a_c, a_s, \mathcal{S}'] = \mathcal{S}.pop() \quad \neg is\_symbolic(a_s) \quad \mathcal{T}_c \vdash a_c : V_c \quad \mathcal{T}_s \vdash a_c : V_s \quad V_s \vdash a_s : s \\ \mathcal{S}''_c = \mathcal{S}'_c.push(c) \quad \mathcal{S}''_s = \mathcal{S}'_s.push(s) \quad \zeta = \Sigma(pc+1)\end{array}}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{SLOAD} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}'', \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc+1, \zeta}$$

$$\text{SLOAD-C-RN}\;\frac{\begin{array}{c}[a_c, a_s, \mathcal{S}'] = \mathcal{S}.pop() \quad \neg is\_symbolic(a_s) \quad \mathcal{T}_c \vdash a_c : V_c \quad \mathcal{T}_s \vdash a_c :\perp \quad s = mk\_symbolic() \\ \mathcal{T}'_s = [a_c : \{a_s \to s\}]\mathcal{T}_s \quad \mathcal{S}''_c = \mathcal{S}'_c.push(c) \quad \mathcal{S}''_s = \mathcal{S}'_s.push(s) \quad \zeta = \Sigma(pc+1)\end{array}}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{SLOAD} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}'', \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc+1, \zeta}$$

$$\text{SLOAD-S-RN}\;\frac{\begin{array}{c}[a_c, a_s, \mathcal{S}'] = \mathcal{S}.pop() \quad is\_symbolic(a_s) \quad \mathcal{T}_c \vdash a_c : V_c \quad \mathcal{T}_s \vdash a_c : V_s \quad \forall (k \to v) \in V_s,\ sat(a_s \neq k) \\ s = mk\_symbolic() \quad V'_s = [a_s : s]V_s \quad \mathcal{T}'_s = [a_c : V'_s]\mathcal{T}_s \quad \mathcal{S}''_c = \mathcal{S}'_c.push(c) \quad \mathcal{S}''_s = \mathcal{S}'_s.push(s) \quad \zeta = \Sigma(pc+1)\end{array}}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{SLOAD} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}'', \mathcal{M}, \mathcal{T}', \mathcal{P}, \mathcal{E}, pc+1, \zeta}$$

$$\text{SLOAD-S-RR}\;\frac{\begin{array}{c}[a_c, a_s, \mathcal{S}'] = \mathcal{S}.pop() \quad is\_symbolic(a_s) \quad \mathcal{T}_c \vdash a_c : V_c \quad \mathcal{T}_s \vdash a_c : V_s \quad \exists (k \to v) \in V_s,\ unsat(a_s \neq k) \\ V_c \vdash a_s : c \quad \mathcal{S}''_c = \mathcal{S}'_c.push(c) \quad \mathcal{S}''_s = \mathcal{S}'_s.push(v) \quad \zeta = \Sigma(pc+1)\end{array}}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{SLOAD} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}'', \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc+1, \zeta}$$

$$\text{SHA3-C}\;\frac{\begin{array}{c}[(a, l_c), \_, \mathcal{S}'] = \mathcal{S}.pop(2) \quad n = \lceil \frac{l_c}{32} \rceil \quad \forall i \in [0, n), \neg is\_symbolic(\mathcal{M}_s(a_i)) \\ h = sha3(a, l_c) \quad \mathcal{S}'' = \mathcal{S}'.push(h) \quad \zeta = \Sigma(pc+1)\end{array}}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{SHA3} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}'', \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc+1, \zeta}$$

$$\text{SHA3-S}\;\frac{\begin{array}{c}[(a, l_c), \_, \mathcal{S}'] = \mathcal{S}.pop(2) \quad n = \lceil \frac{l_c}{32} \rceil \quad \exists j \in [0, n), is\_symbolic(\mathcal{M}_s(a_j)) \quad h_c = sha3(a, l_c) \quad \mathcal{S}''_c = \mathcal{S}'_c.push(h_c) \\ h_s = concat(\mathcal{M}_s(a_0), ..., \mathcal{M}_s(a_{n-1})) \quad \mathcal{S}''_s = \mathcal{S}'_s.push(h_s) \quad \zeta = \Sigma(pc+1)\end{array}}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, \texttt{SHA3} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}'', \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc+1, \zeta}$$

Fig. 8. Operational semantics for the storage accesses, involving the instructions `SLOAD`, `SSTORE` and `SHA3`.

$$\text{MT}\;\frac{end\_of\_path(inst) \quad \zeta = \Sigma(0)}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, pc, inst \Rightarrow \Sigma, \emptyset, \emptyset, \emptyset, \mathcal{T}, \mathcal{P}, \mathcal{E}, 0, \zeta}$$

Fig. 9. Semantics for a multi-transactional execution.

the framework can restart a new execution (i.e., transaction) by preserving the storage state and the path condition, such that the new transaction can follow current execution and reuse the changed state of the smart contract.

4 Implementation

We implement the DSE framework on top of `aleth` [19], the C++ implementation of Ethereum. We choose Z3 [18] as the SMT solver and encode the symbolic values in 256-bit bitvectors. We set the array relevant restrictions in Figure 5 as $\mathbb{X} = 100$ and $\mathbb{Y} = 1000$. In addition, we construct appropriate inputs that can trigger specific paths according to the results of constraint solving and the learned information for arrays.

Nova. To demonstrate the effectiveness of the DSE framework, we implement Nova to detect one of the most harmful vulnerabilities, the integer overflow. To perform the detection, Nova maintains a new state $\mathcal{O}$ for *overflow* constraints corresponding to suspicious arithmetic operations, e.g., `ADD`. An overflow constraint tells about what condition can be satisfied if an overflow occurs. The ADD rule in Figure 10 gives an example. We save in $\mathcal{O}$ the expression "$s < p$". When a vulnerability is reported, Nova generates the exploitable inputs, helping analysts verify the vulnerability.

Delayed Checking. Given the fact that smart contracts may terminate abnormally and revert the state, checking the overflow issues immediately for suspicious operations can result in tremendous false alarms. Nova adopts a mechanism of delayed checking, detecting the vulnerabilities only at the end of a normal termination. RETURN-CHECK in Figure 10 iteratively checks if any overflow constraint is satisfactory with known conditions $\mathcal{P}$ and $\mathcal{E}$. If Z3 can find a solution, Nova reports the vulnerability. Note that this rule is applied before the path exploration rules in Figure 4.

Storage-write oriented. In some cases, integer overflows cannot result in any loss to the users even if the corresponding path terminates normally, e.g., a user-provided overflow check in Megawttcoin [20] as shown below. Therefore, Nova examines only the paths that write to the storage.

```
if (balances[msg.sender] + _value <=
    balances[msg.sender])
   return false;
```

Mtvd. We have also experimentally implemented Mtvd, in order to demonstrate the capability of the DSE framework for detecting multi-transactional vulnerabilities such as ether leaking and suicidal [21], [22]. While vulnerabilities across multiple transactions are difficult to detect without sufficient knowledge about the vulnerability logics, we follow the heuristics presented in [21], [22]. If a specific user (a.k.a, *attacker*) is able to make money from a contract but he has never sent any ethers to the contract, we report a leaking vulnerability in the contract. If an attacker can kill a contract by executing a specific inter-transactional path and bypassing the safety checks before the destruction operation, we report a suicidal vulnerability.

Different with Nova which treats all storage data as arbitrary and does not differentiates the contract users, Mtvd considers the user (i.e., `msg.sender`) as a potential attacker

$$\text{ADD}\ \frac{[(a,b),(p,q),\mathcal{S}'] = \mathcal{S}.pop(2) \quad c = a+b \quad \mathcal{S}''_c = \mathcal{S}'_c.push(c) \quad s = p+q \quad \mathcal{S}''_s = \mathcal{S}'_s.push(s) \\ o = s < p \quad \mathcal{O}' = is\_symbolic(s)?\mathcal{O}.push(o) : \mathcal{O} \quad \zeta = \Sigma(pc+1)}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, \mathcal{O}, pc, \texttt{ADD} \Rightarrow \Sigma, \mathcal{C}, \mathcal{S}'', \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, \mathcal{O}', pc+1, \zeta}$$

$$\text{RETURN-CHECK}\ \frac{not\_empty(\mathcal{O}) \quad [o,\mathcal{O}'] = \mathcal{O}.pop() \quad L = solve(\mathcal{P} \wedge \mathcal{E} \wedge o) \quad L \neq \emptyset ? report\_vulnerability(o,L) : \perp}{\Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, \mathcal{O}, pc, \texttt{RETURN} \Rightarrow (\texttt{RETURN-CHECK}\ only)\ \Sigma, \mathcal{C}, \mathcal{S}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{E}, \mathcal{O}', pc, \texttt{RETURN}}$$

Fig. 10. Semantics for collecting overflow constraints at suspicious instructions (e.g., `ADD`) and delayed checking at `RETURN`.

and assigns it a specific value 0xFF...FF. Any correlated storage slots are initially empty, e.g., `balances[msg.sender] = 0`. Under such settings, MTVD first performs symbolic executions on all possible transactions and discovers paired write/read of the same storage slots across transactions. We consider the paired write/read operations can be related to bypass the safety checks, e.g., `owner = msg.sender` in one transaction and `if (owner == msg.sender)` in the other transaction. After successfully executing the first transaction, we use the storage state as the initial state of executing the second transaction. In other words, the first access to some storage data (e.g., `owner` in the above example) in the second transaction will meet a concrete value instead of a symbolic state. By this means, if the attacker successfully triggers the sensitive operations, MTVD will report the vulnerabilities and corresponding exploitable inputs for verification.

5 Evaluation

In this section, we will present the evaluation results for the smart contracts deployed on Ethereum and discuss the causes of the inaccuracies and the comparisons with two state-of-the-art bug-finding tools.

5.1 Experimental Setup

To ease the process of auditing the evaluation results, we collected 21,016 open-source smart contracts from the Etherscan website [23] and their corresponding bytecode in the blockchain.

We evaluated NOVA on all the collected smart contracts on a machine with Intel Xeon Silver 4110 CPU and 82 GB memory with Ubuntu 18.04. We parallelize four NOVA instances spontaneously to speed up the analysis. Because Z3 may be stuck with certain complex constraints, we set the maximum time for solving a given constraint to 100 seconds. Timeout is treated as *unsolvable*, i.e., there do not exist any inputs that satisfy the constraint. While most smart contracts can be finished quickly, some cost hours as we will see in Section 5.2. Hence, we set the maximum time for one smart contract to five minutes.

To show the effectiveness of NOVA, we compare it with four state-of-the-art tools, OYENTE [9], MYTHRIL [10], OSIRIS [11] and VERISMART [12].

To evaluate the effectiveness of MTVD, we first run it on the vulnerable smart contracts mentioned in [21], [22], and then perform the detection on all the contracts from our repository.

5.2 Experiment Results of NOVA

NOVA reports 16,108 integer overflow vulnerabilities in total. Verifying the correctness of the vulnerabilities can be

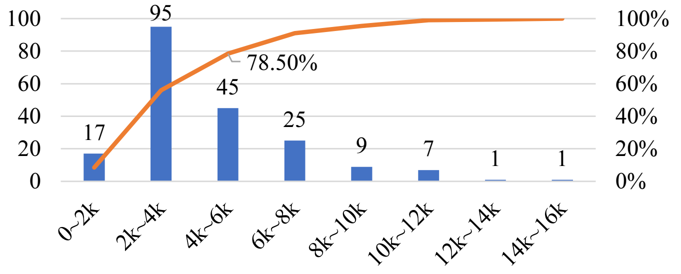


Fig. 11. Summary of the 200 smart contracts. The X-axis shows the size of the smart contracts in the form of the number of bytecode instructions. The bars denote the number of smart contracts of the corresponding sizes and the line represents the accumulated percentage.

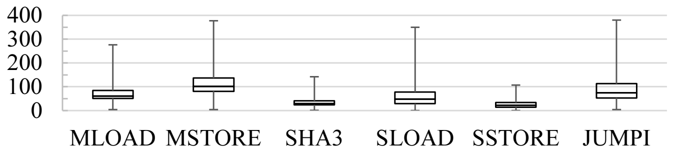


Fig. 12. Statistics of the instructions for memory/storage accesses, hash and branch operations in the 200 smart contracts.

done in a private blockchain and we use Remix [8] for this purpose. Because the occurrence of an integer overflow will not crash the execution, we feed the smart contracts with the generated exploits and inspect the storage changes to confirm the occurrences of the vulnerabilities.

While verifying all the reported vulnerabilities is time-consuming and tedious and we do not have an oracle of the contracts, we focus those contracts reported by at least two tools to contain integer overflow vulnerabilities. Since all other tools have reported many vulnerabilities that are not covered by NOVA, we get a chance of inspecting whether and why NOVA misses some real problems. we randomly selected 200 smart contracts for inspection.

Scalability. Figure 11 presents the size of the 200 smart contracts (i.e., the number of instructions) and the accumulated distribution, and Figure 12 shows the statistics of the instructions for memory/storage accesses, hash and branch operations. Though the smallest contract contains 182 instructions, the average size is 4,443 and more than 3/4 of the smart contracts contain fewer than 6,000 instructions. It is also noticeable that NOVA can scale to large contracts with more than 10k instructions. In fact, it completes the analysis for the largest contract (15,873 instructions, more than 800 memory/storage accesses and 380 branches) in 103 seconds and reports three real overflows.

Performance. Figure 13 presents the number of paths NOVA has executed for each smart contract and the corresponding time cost. On average, NOVA executes about 44 paths per smart contract. While the number of paths is displayed in an ascending order, the time cost is not linear,

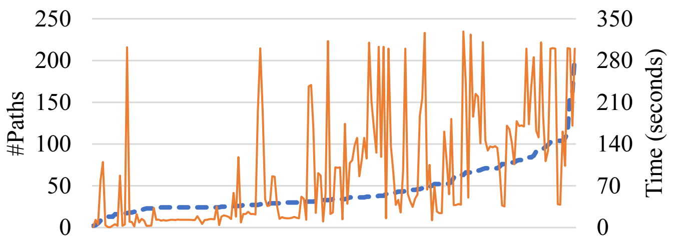


Fig. 13. The number of paths NOVA has executed (dashed line, in an ascending order) and the corresponding time cost (solid line) for the 200 smart contracts.

mainly because of the cost for constraint solving. Among the 200 smart contracts, 20 cost more than five minutes. For the other 180 smart contracts, the average time cost is 70.2 seconds, demonstrating the efficiency of NOVA.

Accuracy. NOVA totally reports 496 vulnerabilities. For each smart contract, we inspect the source code and the reports from the other tools. Besides, we exploit the reported vulnerabilities with the generated exploits in Remix. 467 vulnerabilities are verified to be true, leaving 29 false warnings. We also discover 35 missed issues. The false positive (FP) rate and false negative (FN) rate are 6.8% and 7.0%, respectively. Below we will discuss the inaccuracies in details.

5.2.1 *False Positives*

We find four FPs in 29 smart contracts. The root cause is that NOVA detects vulnerabilities per path and some semantic *restriction* information from another path is not taken into account. The EdgelessCasino [24] contract presents an example as below.

```
1  contract CasinoBank {
2    uint public waitingTime;
3    function setWaitingTime(uint nwt) ... {
4      require(nwt <= 24 hours);
5      waitingTime = nwt;
6    }
7    function requestWithdrawal() public {
8      withdrawAfter[msg.sender] = now + waitingTime;
9    }
10 }
```

NOVA reports an integer overflow at line 8. Looking at only that line, a constant (`now`) plus a field variable is very likely to overflow, when the field variable `waitingTime` is modifiable elsewhere. However, `waitingTime` is initially set to 90 minutes and when the owner intends to modify it, it is restricted to be at most 24 hours at line 4. Consequently, the *vulnerable* addition acquires a time value in near future. Given the fact that the time can be accurately represented in traditional 64-bit machines, the addition will never overflow in 256-bit Ethereum. Therefore, we consider it as a FP.

SBGToken [25] shows a more typical case of false positives. from only the function `burn`, NOVA reports a subtraction overflow at line 4 as the operation is not protected as the other subtraction at line 3. However, in the smart contract, `totalSupply` denotes the total number of tokens held by the owner and all investors, whose accounts are maintained by `balanceOf`. As a result, when the constraint at line 2 holds, `totalSupply` will never be smaller than `value`. Hence, line 4 will not result in any overflow and NOVA emits a FP here.

```
1 function burn(uint256 _value) public ... {
2    require(balanceOf[msg.sender] >= _value);
3    balanceOf[msg.sender] -= _value;
4    totalSupply -= _value;
5 }
```

Suppressing such FPs is nontrivial without the knowledge about the semantic restrictions and particularly difficult if we only have the deployed bytecode at hand. Automatically inferring the semantic restrictions usually requires cross-path analysis, which deeply inspects the whole program and builds the relations among different operations to the same data (for EdgelessCasino), or even a thorough understanding of the source code (for SBGToken). However, too heavyweight analysis or manual intervention hinders the applicability for large-scale integer overflow detection. Fortunately, the small number of such cases makes it reasonable to keep away from those solutions. Another way of handling the FP issues could be to execute the smart contracts with user inputs as the initial storage states, so as to avoid the interference of impractical arbitrary initial storage.

5.2.2 *False Negatives*

NOVA misses 35 integer overflows. We carefully examined them and found the constraint solving is responsible for the FNs.

Safe multiplication (*safemul*) is an obstacle for Z3 to solve the constraints. When a safemul is presented in a path, the path constraint introduced by `assert` is collected, e.g., `(a == 0 || c / a == b)`. With this constraint, Z3 fails to solve the final constraints for vulnerability detection. Below is an example in MainSale [26].

```
1 contract MainSale {
2   uint public commandPercent = 10;
3   function mintTokensForCommand(address recipient,
        uint tokens) ... {
4     max = token.totalSupply().mul(commandPercent)
5              .div(100 - commandPercent)
6              .div(1 ether);
7   }
8 }
```

The field `commandPercent` is supposed to be bounded but it is modifiable in another function. The adversaries may modify `commandPercent` or an incidental operation may put an unexpected value for it, leading to an overflow at line 5. The use of safe math library introduces the safemul assertion into the path condition. Z3 cannot tell the result before timeout.

We further examine Z3's capability of solving constraints with the safemul assertion with different bitvector lengths. When we encode the integer values to 32 or 64 bits, the same constraints can be solved in one second. Even if the integers are 255-bit long, Z3 completes the solving in several minutes on a typical laptop. However, 256-bit encoding prevents it from solving the corresponding constraint even for hours. Improving Z3 could help but it is beyond the scope of this paper.

Timeout makes the contribution to 27 FNs. NOVA involves a lot of symbolic comparisons to determine a storage

TABLE 1
The number of total paths, time cost and newly discovered vulnerabilities for two smart contracts finished within one day.

	#Paths	Time (hours)	#NewVuln
MainSale [26]	123	5.05	3
POSC [27]	115	8.35	4

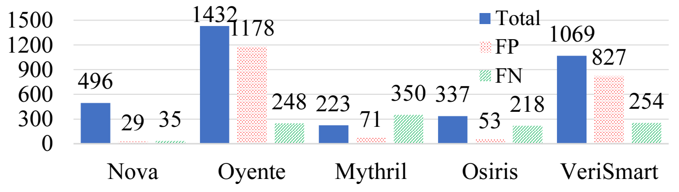


Fig. 14. Comparison results for the 200 smart contracts among NOVA and four state-of-the-art tools, including the total number of reported vulnerabilities, the number of false alarms and false negatives.

access and each comparison can cost massive time. Besides, any involved safemul can lead to timeout of the solver. Hence, the maximum time of five minutes can be easily exhausted with abundant symbolic comparisons and safemul operations.

We further examine the FNs. They are distributed in five different smart contracts, among which one has 6k+ instructions, one has 10k+ instructions and the rest contain 9k+ instructions.

We re-run NOVA on the five smart contracts with such FNs for at most 24 hours. Seven missed flaws (7/27 = 25.9%) in two smart contracts are uncovered. Table 1 lists the results. MainSale [26] takes more than five hours to complete and POSC [27] costs more than eight hours. Unfortunately, the other three smart contracts cannot be finished within 24 hours and we terminate their executions without any issues reported.

A possible solution can be to leverage static analysis to prioritize the paths. Simple paths with fewer time-consuming operations are executed first such that potential flaws can be discovered with a given time budget.

5.3 NOVA v.s. State-of-the-art Tools

We compare NOVA with OYENTE, MYTHRIL, OSIRIS and VERISMART on the 200 smart contracts with the same timeout settings, i.e., five minutes for one smart contract. Figure 14 shows the comparison results. Compared with NOVA, OYENTE and VERISMART report much more vulnerabilities with a lot of FPs while MYTHRIL and OSIRIS detect fewer flaws but misses hundreds of real problems.

Table 2 presents the detection accuracy of the five tools, in which $Precision = \frac{\#TP}{\#TP+\#FP}$ and $Recall = \frac{\#TP}{\#TP+\#FN}$, where $\#TP = \#Total - \#FP$. We can see that NOVA presents much higher precision and recall. Among the four tools, OSIRIS, with significant improvements on top of OYENTE, gains the highest accuracy but still shows lower precision and recall than NOVA.

We will present in next section several typical cases that are successfully detected by NOVA but inaccurately handled by two tools under active development.

TABLE 2
Detection accuracy of NOVA and state-of-the-art tools.

	NOVA	OYENTE	MYTHRIL	OSIRIS	VERISMART
Precision	94.2%	17.7%	68.2%	84.3%	22.6%
Recall	93.0%	50.6%	30.3%	56.6%	48.8%

5.3.1 *Case Studies*

Case 1: multiplication. OYENTE does not support the detection of multiplication overflows. Below is a very simple function in Klassicoin [28] and the vulnerable multiplication at line 4 is missed by OYENTE. Our DSE framework treats the user account address `msg.sender` as a special input and assigns it a constant value. By solving the constraint $msg.sender == fundsWallet$, it gives the field `fundsWallet` a proper value that is reused in the second execution by the rule SLOAD-C-RN in Figure 8, such that the concrete execution can reach line 4. NOVA builds an overflow constraint for the operation and checks vulnerabilities with RETURN-CHECK in Figure 4 when the execution reaches the end.

```
1 function changeSupply(uint supp) returns (uint) {
2    if (msg.sender != fundsWallet)
3       throw;
4    totalSupply = supp * 1000000000000000000;
5    return totalSupply;
6 }
```

Case 2: array operations. Arrays in the input usually involve all the data regions, as we have discussed in Section 3. Besides, some smart contracts contain array-typed fields. Failing to precisely handle the arrays is a major cause of the inaccuracies. OYENTE skips the instruction CALLDATACOPY and ignores any connections between the input array and the memory. As a result, it lacks the ability of detecting array related problems, e.g., the potential addition overflow in Figure 1. MYTHRIL, on the contrary, is able to handle the first array in the input but fails for additional arrays. RocketCoin [15] contains a function as below.

```
1 function multiTransfer(address[] _addresses,
      uint[] _amounts) ... {
2   uint totalAmount;
3   for (uint a = 0; a < _amounts.length; a++) {
4     totalAmount += _amounts[a];
5   } ... // more operations are omitted
6 }
```

The addition at line 4 operates the second array in the input, which exceeds the capability of OYENTE and MYTHRIL, causing an undetected vulnerability. NOVA treats every array in the same manner and can successfully report the addition overflow vulnerability in this smart contract.

OYENTE and MYTHRIL also report a great number of false warnings related to the array calculation. The array metadata (the length and offset) in the input is normally associated with a symbolic variable. Such values can involve in arithmetic operations, e.g., adding a constant to the offset for the actual position of the array and multiplying 0x20 to get the byte size of the array for memcopy. These arithmetic operations can be detected as vulnerabilities. However, as we discussed in Section 3.3, it is impractical to feed the smart contract with a very large array. Our DSE framework imposes restrictions to the array metadata as done in Figure 5, NOVA does not report corresponding calculations as potential overflows. The FP emitted by MYTHRIL for Figure 1 is related to this case. OYENTE does not report the multiplication overflow, but it still reports a possible problem if the length of `to` is 2^{256}, which is actually impossible. Similarly, it produces a lot of false alarms when performing arithmetic operations on the sizes or offsets of the storage fields of array types, e.g., `string`, `uint[]`.

5.3.2 *Orthogonality*

From the comparison, we can see that our approach as well as NOVA can complement existing detection tools to hit the vulnerabilities that are missed by other tools. Besides, we believe that NOVA can be leverated to validate the detection results of static analysis tools.

5.4 Experimental Results of MTVD

As mentioned earlier, we evaluate MTVD on the vulnerable contracts present in [21], [22] and all the selected contracts. MTVD successfully identifies the leaking and suicidal vulnerabilities reported in previous works. Besides, it reports three ether leaking vulnerabilities and one suicidal vulnerability, without false warnings.

An example is Zemana [29] shown below, in which the function `QuantumPay` is publicly accessed. An attacker can invoke it, changing the contract owner at line 3, and then calls `withdraw` to steal the ethers in the smart contract at line 9, without investing any ethers to the contract.

```
1  contract Zemana is ERC20 {
2     function QuantumPay () public {
3        owner = msg.sender;
4        distr(owner, totalDistributed);
5     }
6     function withdraw() onlyOwner public {
7        address myAddress = this;
8        uint256 etherBalance = myAddress.balance;
9         owner.transfer(etherBalance);
10    }
11 }
```

5.4.1 *Discussion*

Detecting multi-transactional vulnerabilities involving deep states is challenging when the states grow exponentially [22]. While the main purpose of this paper is to present a DSE framework for smart contracts, we did not pay much attention to a detector that is capable to detect arbitrary multi-transactional vulnerabilities. Instead, we implement MTVD with naive heuristics, to simply demonstrate that we can extend the DSE framework to detect such vulnerabilities. However, we believe the other researchers with insightful ideas can also leverage the DSE framework to implement scalable and effective detectors.

6 RELATED WORK

In this section, we will review the research efforts on smart contracts. Leveraging symbolic execution techniques accounts for a major part of the effort. Some researchers transform the smart contracts, either source code or bytecode, to another intermediate language and verify if certain specifications are violated. In addition, dynamic testing techniques are applied to discover the vulnerabilities.

OYENTE [13] is one of the earliest effort to detect vulnerabilities in smart contracts with the help of symbolic execution. The maintainers also augmented OYENTE for integer overflow vulnerability detection [9]. MANTICORE [30], MYTHRIL [10] and OSIRIS [11] works in similar way as OYENTE to detect the vulnerabilities. TEETHER [31] searches critical instructions and generates exploits for the corresponding paths by utilizing the symbolic execution techniques. TEETHER handles the hash operations by evaluating the hash's inputs to obtain concrete values, over which the hash value is computed. Our approach hashes on concrete values too, but the concrete values are assigned at the first access and propagated along the execution. Our method reduces the expense of constraint solving at every hash operation, compared with TEETHER. SCOMPILE [32] identifies the critical paths involving monetary transactions and only apply symbolic execution techniques to top ranked critical paths. With similar consideration, NOVA examines only the paths with storage modification. MAIAN [21] employs static symbolic analysis to find the properties of execution traces that are across multiple invocations of a smart contract, in order to detect the greedy, prodigal and suicidal contracts. This idea can be helpful to suppress some false warnings we have mentioned in Section 5.2.1. ETHBMC [14] models all the data accesses fully symbolically while our approach employs concrete execution to easily determine the nonuniform data accesses.

SECURIFY [33] does not directly employ symbolic execution to uncover the security issues. Instead, it extracts semantic information through symbolic analysis and then checks if any predefined safety properties could be violated. Similarly, VANDAL [34] decompiles the bytecode into a language that can expose the data- and control-flow structures, and then inspect if the smart contract violates any specifications. SMARTCHECK [35] converts the source code of smart contracts into an XML format and detects vulnerability patterns by using XPath queries on the XML. ZEUS [36] translates the Solidity source code into LLVM [37] intermediate representations and leverage the LLVM framework to perform abstract interpretation and symbolic model checking. Hildenbrandt et al. describe the EVM semantics based on the $\mathbb{K}$ framework [38] to verify important properties [39]. VERISMART [12] applies a domain-specific algorithm to discover the transaction invariants and leverage them to verify the smart contracts source code. VERX [40] formalizes the temporal safety properties, instruments the smart contracts and then verifies the functional properties. VERX models the storage in the theory of arrays and replaces the original `SHA3` operation with a simpler Z3 function which constraints the behavior with weak assumptions. The new hash function maps the input to the hash result and can also be used in the DSE setting. In addition, we think that the replacement of the hash function can be extremely useful, compared with `SHA3`, when we aim to restore a reasonable input for a specific hash value (see the example in [14]). Jiao et al. develop a formal semantics for Solidity and verifies the smart contracts against the semantics-level security properties [41]. Compared with the above approaches, we are the first to describe the semantics suitable for dynamic symbolic execution. Moreover, the applicability of the approaches requiring the source code is limited as only a very small portion of the smart contracts are open-sourced [21], [42].

Annotary [43] performs concolic execution with the help of developer-written annotations in Solidity source code and we think the solution can also be helpful in address the FP issues. SEREUM [44] performs taint analysis and monitors the executions to protect the smart contracts against the reentrancy bugs. CONTRACTFUZZER [45] leverages the fuzzing techniques to test the smart contracts for vulnerability detection by analyzing the runtime behaviors. Grossman et al. proposed an online method to identify the callback free objects for detecting the reentrancy vulnerabilities [46]. ETHRACER [47] leverages dynamic symbolic execution and testing techniques to detect the order sensitive event relevant bugs. TOKENSCOPE [48] monitors the behaviors inconsistent with standard token interfaces. ILF [22] includes neural networks to learn a proper fuzzing policy from the generated inputs of symbolically analyzed smart contracts and uses the policy to fuzz new programs. Based on the heuristics provided by ILF, we build MTVD to detect two kinds of multi-transactional vulnerabilities, i.e., ether leaking and suicidal.

7 CONCLUSION

With the proliferation of cryptocurrencies and the smart contracts, the security aspect has aroused great interest. As a successful technique in traditional platform, dynamic symbolic execution (DSE) is promising to detect vulnerabilities in smart contracts. However, simply employing the DSE solutions from other platforms is challenging due to the nonuniform data access nature in smart contracts. In fact, smart contracts compromise multiple addressing modes, including typical flat address mode and complicated key-value mode. In this paper, we propose a DSE framework that is described in operational semantics. The framework resolves the nonuniform data access issues precisely and comprehensively by presenting proper solutions for different addressing modes. On top of the framework, we implement NOVA to detect the integer overflow flaws, one of the most harmful vulnerabilities that have caused substantially financial loss. We evaluate NOVA on real-world smart contracts and verify the reported vulnerabilities in a private blockchain. Comparisons demonstrate that NOVA outperforms state-of-the-art bug-finding tools and the results show the effectiveness of our approach, with high precision and recall over 90%. We also implement MTVD to detect two kinds of multi-transactional vulnerabilities and discover four problems in real-world smart contracts.

ACKNOWLEDGMENTS

This work is supported in part by National Natural Science Foundation of China (NSFC) under grants U1836209 and 61802413, the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China under grants 19XNLG02 and 20XNLG03.

REFERENCES

[1] K. Finley, "A $50 million hack just showed that the dao was all too human," Jun. 2016. [Online]. Available: https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/

[2] T. McCallum, "Exploding rockets & millions of free tokens? let's take a good look at integer overflows," Feb. 2019. [Online]. Available: https://hackernoon.com/exploding-rockets-millions-of-free-tokens-lets-take-a-good-look-at-integer-overflows-2800794e48d9

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2006, pp. 322–335.

[4] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proc. 38th Int. Conf. Softw. Eng.*, May 2016, pp. 144–155.

[5] B. Loring, D. Mitchell, and J. Kinder, "Sound regular expression semantics for dynamic symbolic execution of javascript," in *Proc. 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Jun. 2019, pp. 425–438.

[6] H. van der Merwe, "Verification of Android applications," in *Proc. 37th Int. Conf. Softw. Eng. - Volume 2*, May 2015, pp. 931–934.

[7] S. Marx, "Understanding ethereum smart contract storage," Mar. 2018. [Online]. Available: https://programtheblockchain.com/posts/2018/03/09/understanding-ethereum-smart-contract-storage

[8] The Ethereum Community, "Remix - ethereum IDE," Accessed: Jun. 2020. [Online]. Available: http://remix.ethereum.org/

[9] "Oyente," Accessed: Jun. 2020. [Online]. Available: https://github.com/melonproject/oyente

[10] "Mythril," Accessed: Jun. 2020. [Online]. Available: https://github.com/ConsenSys/mythril

[11] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 664–676.

[12] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *Proc. IEEE Symp. Secur. Privacy*, May 2020, pp. 1678–1694.

[13] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 254–269.

[14] J. Frank, C. Aschermann, and T. Holz, "EthBMC: A bounded model checker for smart contracts," in *Proc. 29th USENIX Secur. Symp.*, Aug. 2020, pp. 2757–2774.

[15] "RocketCoin," Accessed: Jun. 2020. [Online]. Available: https://etherscan.io/address/0x6fc9c554c2363805673f18b3a2b1912cce8bfb8a

[16] B. Meyer, "Ethereum smart contract vulnerabilities can lead to millions in losses," Nov. 2020. [Online]. Available: https://cybernews.com/security/ethereum-smart-contract-vulnerabilities

[17] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proc. 29th ACM SIGSOFT Int. Symp. on Software Testing and Analysis*, Jul. 2020, pp. 415–427.

[18] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proc. of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[19] "Aleth - Ethereum C++ client, tools and libraries," Accessed: Jun. 2020. [Online]. Available: https://github.com/ethereum/aleth

[20] "Megawttcoin," Accessed: Jun. 2020. [Online]. Available: https://etherscan.io/address/0x7f1f1e95b243708aa29cfba53e13d45e28356d2b

[21] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. 34th Annu. Comput. Secur. Appl. Conf*, Dec. 2018, pp. 653–663.

[22] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 531–548.

[23] Etherscan, "Ethereum (ETH) blockchain explorer," Accessed: Jun. 2020. [Online]. Available: https://etherscan.io/

[24] "Edgelesscasino," Accessed: Jun. 2020. [Online]. Available: https://etherscan.io/address/0x15f08079310e2c9dacaa73c0e450368185724aea

[25] "SBGToken," Accessed: Jun. 2020. [Online]. Available: https://etherscan.io/address/0xb5980eb165cbbe3809e1680ef05c3878ce25dacb

[26] "MainSale," Accessed: Jun. 2020. [Online]. Available: https://etherscan.io/address/0xb2819a0c3db0b9513a5ddd747f873877f622e083

[27] "POSC," Accessed: Jun. 2020. [Online]. Available: https://etherscan.io/address/0x3d807baa0342b748ec59aa0b01e93f774672f7ac

[28] "Klassicoin," Accessed: Jun. 2020. [Online]. Available: https://etherscan.io/address/0xb85815bd8f610867c0b59dd386ca2b0609fa0942

[29] "Zemana," Accessed: Jun. 2020. [Online]. Available: https://etherscan.io/address/0x63e89a05a3185100aa05eae9b5e15b00f4a1687d

[30] 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 *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2019, pp. 1186–1189.

[31] J. Krupp and C. Rossow, "teEther: Gnawing at ethereum to automatically exploit smart contracts," in *Proc. 27th USENIX Secur. Symp.*, Aug. 2018, pp. 1317–1333.

[32] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "sCompile: Critical path identification and analysis for smart contracts," in *Formal Methods and Software Engineering*. Cham, Switzerland: Springer, 2019, pp. 286–304.

[33] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 67–82.

[34] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *CoRR*, vol. abs/1809.03981, pp. 1–28, Sep. 2018. [Online]. Available: http://arxiv.org/abs/1809.03981

[35] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proc. 1st Int. Workshop on Emerging Trends in Software Engineering for Blockchain*, May 2018, pp. 9–16.

[36] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: analyzing safety of smart contracts," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2018, pp. 1–15.

[37] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, Mar. 2004, pp. 75–86.

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

[39] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "KEVM: A complete formal semantics of the ethereum virtual machine," in *31st IEEE Computer Security Foundations Symposium*, 2018, pp. 204–217.

[40] A. PERMENEV, D. DIMITROV, P. TSANKOV, D. DRACHSLER-COHEN, and M. VECHEV, "Verx: Safety verification of smart contracts," in *Proc. IEEE Symp. Secur. Privacy*, May 2020, pp. 1661–1677.

[41] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of solidity," in *Proc. IEEE Symp. Secur. Privacy*, May 2020, pp. 1695–1712.

[42] M. Fröwis and R. Böhme, "In code we trust? - measuring the control flow immutability of all smart contracts deployed on ethereum," in *International Workshops on Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Cham, Switzerland: Springer, 2017.

[43] K. Weiss and J. Schütte, "Annotary: A concolic execution system for developing secure smart contracts," in *Computer Security – ESORICS 2019*. Cham: Springer, Sep. 2019, pp. 747–766.

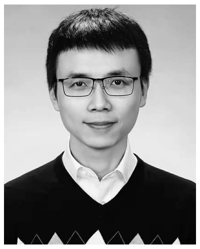
[44] M. Rodler, W. Li, G. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *Proc. 26th Annu. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2019, pp. 1–15.

[45] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, Sep. 2018, pp. 259–269.

[46] 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," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017.

[47] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing and Analysis*, Jul. 2019, pp. 363–373.

[48] 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 *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 1503–1520.



Jianjun Huang received the Ph.D. degree in Computer Science from Purdue University. He is currently an assistant professor at School of Information, Renmin University of China. His research interests focus on program analysis, vulnerability detection, mobile security and blockchain security.

Jiasheng Jiang received the B.S. degree in Information Security from Renmin University of China. He is currently a graduate student at School of Information, Renmin University of China. His research interests focus on blockchain security.

Wei You received the Ph.D. degree in Computer Science from School of Information, Renmin University of China. He is currently an associate professor at School of Information, Renmin University of China. His research interests focus on program analysis, mobile security and Web security.

Bin Liang received the Ph.D. degree in Computer Science from Institute of Software, Chinese Academy of Sciences. He is currently a professor at School of Information, Renmin University of China. His research interests focus on program analysis, vulnerability detection, mobile security and AI security.