ARTIFACT EVALUATED usenix ASSOCIATION PASSED

SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution

Sunbeom So, Seongjoon Hong, Hakjoo Oh*
Department of Computer Science and Engineering
Korea University

Abstract

We present SmarTest, a novel symbolic execution technique for effectively hunting vulnerable transaction sequences in smart contracts. Because smart contracts are stateful programs whose states are altered by transactions, diagnosing and understanding nontrivial vulnerabilities requires generating sequences of transactions that demonstrate the flaws. However, finding such vulnerable transaction sequences is challenging as the number of possible combinations of transactions is intractably large. As a result, most existing tools for smart contract analysis use abstractions and merely point out the locations of vulnerabilities, which in turn imposes a steep burden on users of understanding the bugs, or have limited power in generating transaction sequences. In this paper, we aim to overcome this challenge by combining symbolic execution with a language model for vulnerable transaction sequences, so that symbolic execution effectively prioritizes program paths that are likely to reveal vulnerabilities. Experimental results with real-world smart contracts show that SmarTest significantly outperforms existing tools by finding more vulnerable transaction sequences including critical zero-day vulnerabilities.

1 Introduction

Securing smart contracts is a pressing issue waiting to be addressed for the upcoming blockchain era. Blockchain is a ground-breaking technology that enables automatic fulfillment of agreed obligations between untrusted parties. The obligations are written in smart contracts, computer programs running on blockchain whose executions are therefore guaranteed to be faithful. Smart contracts are gaining popularity across diverse application domains where security and privacy are important [29]. Unfortunately, however, the safety of smart contracts itself remains a major concern. Smart contracts are attractive targets for attackers since they typically manipulate valuable data such as digital assets and therefore even a single glitch can cause tremendous financial damage [1, 5]. Even worse, smart contracts are immutable and their vulnerabilities cannot be mitigated once deployed. Developing techniques to ensure their safety before deployment is critically important and urgent.

In this paper, we present SmarTest, a new safety analyzer for Ethereum smart contracts. The key feature of SmarTest, which differs crucially from existing analyzers [3, 6, 24, 25, 28, 30, 31, 36, 37], is that it effectively finds *vulnerable transaction sequences* of smart contracts. Ethereum smart contracts are stateful programs whose global states are altered by receiving and processing a series of transactions. Therefore, nontrivial bugs in smart contracts are typically caused by the interaction of multiple transactions, and understanding such bugs requires contriving concrete scenarios in terms of transaction sequences. The primary goal of SmarTest is to automate this process; SmarTest aims not only to detect bugs in smart contracts, but to automatically generate vulnerable transaction sequences that prove the flaws.

Existing analyzers for smart contracts fall short in this aspect. For example, existing safety verifiers (e.g. [6, 24, 36]) are fundamentally limited in producing vulnerable transaction sequences because they abstract the set of all transaction sequences into single transaction invariants (i.e. properties that hold under arbitrary interleaving of transactions [36]). Bug-finders such as Oyente [28] and Osiris [37] are only able to indicate certain vulnerable points in smart contracts without generating transaction sequences that reveal vulnerabilities. As a result, triaging vulnerabilities reported by these tools is difficult and error-prone since users need to manually identify concrete scenarios to understand root causes of the vulnerabilities. A few symbolic execution tools (e.g. [3, 25, 30, 31]) support tracing vulnerable transaction sequences but, as we demonstrate in this paper, their performance is far from satisfactory for real-world smart contracts.

To find vulnerable transaction sequences effectively, we present a novel technique that guides symbolic execution with language models. Basically, our technique exhaustively

*Corresponding author: hakjoo_oh@korea.ac.kr

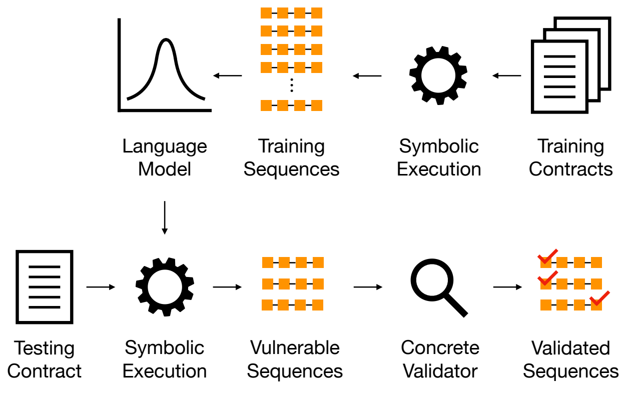


Figure 1: Overview of SMARTEST

enumerates transaction sequences in increasing size and runs symbolic execution over the sequences to decide whether they are vulnerable or not. A main technical challenge that arises in this method is that the number of transaction sequences to be examined grows exponentially as the size of the sequences increases. Our key idea to address this challenge is to guide symbolic execution with statistical language models, so that guided symbolic execution can effectively prioritize transaction sequences that are likely to reveal vulnerabilities. More specifically, given a set of training transaction sequences that are automatically obtained by running unguided symbolic execution on existing vulnerable contracts, our technique automatically learns a probability distribution over vulnerable transaction sequences. Then, symbolic execution guided by the learned model can effectively find vulnerable transaction sequences for new, unseen smart contracts. Figure 1 depicts our approach.

Experimental results show that our language model-guided symbolic execution is highly effective in hunting vulnerable transaction sequences. We implemented SMARTEST for Solidity [4], the most widely used programming language for Ethereum smart contracts, and evaluated it on two datasets with different types of known vulnerabilities. The first dataset is comprised of 443 smart contracts with CVE-reported arithmetic vulnerabilities (e.g., integer overflows). The second dataset consists of 104 contracts with access control-related vulnerabilities, namely Ether-leaking and suicidal contracts [31]. On CVE dataset, we compared SMARTEST with MYTHRIL [3] and MANTICORE [30], two well-known symbolic execution tools developed by blockchain security firms. The results show that SMARTEST found 93.0% of known vulnerabilities out of sampled contracts, whereas MYTHRIL and MANTICORE collectively found 37.2%. On the second dataset with leaking and suicidal contracts, we compared SMARTEST with four symbolic executors and one fuzzer: MAIAN [31], TEETHER [25], MYTHRIL, MANTICORE, and ILF [19]. The results show that SMARTEST effectively found more vulnerabilities than these tools. Moreover, SMARTEST found a number of critical zero-day vulnerabilities from smart contracts in the wild.

```
contract SocialChain {
  uint totalSupply;
  mapping(address=>uint) balance;
  mapping(address=>mapping(address=>uint)) allowance;

  constructor (uint initialSupply) {
    totalSupply = initialSupply;
    balance[msg.sender] = initialSupply;
  }

  function transfer (address to, uint value)
  public returns (bool) {
    require (balance[msg.sender] >= value);
    balance[msg.sender] -= value;
    balance[to] += value;
    return true;
  }

  function approve (address spender, uint value)
  public returns (bool) {
    allowance[msg.sender][spender] = value;
    return true;
  }

  function transferFrom (address from, address to,
        uint value) public returns (bool) {
    require (balance[from] >= value);
    require (balance[to] + value > balance[to]);
    require (allowance[from][msg.sender] >= value);
    balance[from] -= value;
    balance[to] += value;
    allowance[from][msg.sender] += value; // bug
    return true;
  }
}
```

Figure 2: A vulnerable contract (simplified for readability).

Contributions. We summarize our contributions below.

- We present a new technique for effectively finding vulnerable transaction sequences in smart contracts. To our knowledge, our work is the first to use language models to steer symbolic execution towards likely paths.
- We extensively evaluate our technique in comparison with five recently-developed tools [3, 19, 25, 30, 31].
- We make our tool, SMARTEST, and benchmarks publicly available. [1] All experimental results are reproducible.

2 Motivating Examples

In this section, we illustrate SMARTEST with examples.

Example 1. Figure 2 shows a token contract, called SCA.[2] It has three global variables: `totalSupply`, `balance`, and `allowance`. `totalSupply` stores the total amount of issued tokens. `balance` is a mapping from account addresses to token balances. `allowance` is a two-dimensional mapping, which maps approved agents' addresses to token amounts that are allowed to use on behalf of original token holders. For

[1] http://prl.korea.ac.kr/smartest
[2] 0xb75a5e36cc668bc8fe468e8f272cd4a0fd0fd773

example, `allowance[A][B]` indicates the amount of tokens that `A` (i.e. the original token holder) allows `B` (i.e. the agent) to spend.

The constructor at lines 6–9 initializes `totalSupply` and `balance[msg.sender]` (i.e. the balance of the contract creator) with the argument (`initialSupply`). By invoking the `transfer` function, a transaction sender (`msg.sender`) can send `value` tokens to a designated account address (`to`). By invoking `approve`, a token holder (`msg.sender`) can set allowance (`allowance[msg.sender][spender]`) for her agent (`spender`). The `transferFrom` function is similar to `transfer` but tokens are transferred from `from` to `to` by the agent (`msg.sender`) of `from`.

The contract has a critical bug in `trasnferFrom`. A successful transaction must decrease both the sender's balance (`balance[from]`) and the agent's allowance (`allowance[from][msg.sender]`) by the same amount of tokens (`value`). At line 32, however, the allowance is mistakenly increased by `value` (that is, `+=` at line 32 should have been `-=`). This logical flaw in this contract can be found by detecting an integer overflow in the agent's allowance. For example, suppose the contract is deployed by a transaction `constructor (V1)` with `msg.sender` = `A`, and then assume two transactions below are processed in sequence:

1. `approve(B,V2)` with `msg.sender` = `A`
2. `transferFrom(A,C,V3)` with `msg.sender` = `B`

where `A` denotes the contract creator, `B` is the `A`'s agent, `C` is another account address, and `V1`–`V3` are 256-bit integer constants that can trigger the overflow at line 32. For example, assume `V1=0x8800...00`, `V2=0x8100...00`, and `V3=0x7f00...00`. In this case, the remaining allowance after the last transaction must be `0x0200...00` but it ends up with `0x0000...00` due to the overflow. Note that this bug does not manifest itself in a single transaction; to reveal the bug at line 32, `transferFrom` must be invoked with `value > 0`, but a direct invocation to `transferFrom` with `value > 0` will throw an exception due to the guard statement at line 29. Therefore, a *transaction sequence* such as the one shown above is required to trigger and understand the bug.

SMARTEST is able to generate such a vulnerable transaction sequence automatically. It reports the scenario described above with concrete argument values of each transaction and automatically demonstrates that following the scenario indeed causes an integer overflow in a real environment (Figure 1).

A few existing tools (e.g. MYTHRIL [3] and MANTICORE [30]) support generating transaction sequences but they are unsatisfactory; they fail to find a vulnerable sequence for demonstrating the bug at line 32 in this medium-sized contract (404 lines) even after 3 hours. SMARTEST addresses this performance issue of symbolic executors with a novel language model-guided symbolic execution. Other existing tools (e.g., [6,24,28,36,37]) do not help here, too. For example, existing safety verifiers such as SMTCHECKER [6], ZEUS [24], and VERISMART [36] or bug-finders such as OYENTE [28] and OSIRIS [37] do not support producing concrete scenarios; they just point out potentially vulnerable locations without any trace information. As a result, bug triage with these tools is time-consuming and error-prone; users should manually analyze reported warnings to decide whether the warnings are true positives or not and, if they are true, to understand how they happen in what situations. SMARTEST aims to reduce this burden on the tool users.

We remark that, in addition to reporting the flaw at line 32 by overflow detection, SMARTEST can directly report that the `transferFrom` function does not decrease the agent's allowance properly, by producing the same transaction sequence with different argument values. For example, the same sequence with `V2=1` and `V3=1` can demonstrate the logical flaw, where the addition at line 32 does not overflow. SMARTEST supports this feature with rules (Appendix C) for detecting violations of ERC20 standard [2].

```
1  contract Goal {
2    address owner;
3    uint totalSupply;
4    mapping(address=>uint) balance;
5    mapping(address=>mapping(address=>uint)) allowance;
6
7    constructor () public {
8      owner = msg.sender;
9      totalSupply = 0;
10   }
11
12   function mintToken (address target, uint amount)
13   public {
14     require (msg.sender == owner);
15     balance[target] += amount; // overflow
16     totalSupply += amount;     // overflow
17   }
18
19   function approve (address spender, uint value)
20   public returns (bool) {
21     allowance[msg.sender][spender] = value;
22     return true;
23   }
24
25   function burnFrom (address from, uint value)
26   public returns (bool) {
27     require (balance[from] >= value);
28     require (allowance[from][msg.sender] >= value);
29     balance[from] -= value;
30     allowance[from][msg.sender] -= value;
31     totalSupply -= value; // underflow
32     return true;
33   }
34 }
```

Figure 3: A vulnerable contract simplified from Goal contract.

Example 2. Figure 3 shows a simplified version of the Goal token contract.[3] There are four global state variables in the contract, where `owner` denotes the owner of the contract, and `balance`, `totalSupply`, and `allowance` are variables that are similar to those in the previous example (Figure 2).

The constructor sets `totalSupply` to `0` and initializes the owner (`owner`) to be the sender of the initial transaction (`msg.sender`). The function `mintToken` allows `owner` to

[3] 0x7b69b78cc7fee48202c208609ae6d1f78ce42e13

issue a designated amount (`amount`) of tokens. The function `approve` is the same as the one in the previous example. The function `burnFrom` allows the transaction sender (`msg.sender`) to decrease the balance (`balance[from]`) of the original token holder (`from`), where `totalSupply` and `allowance[from][msg.sender]` are equally decreased.

This contract has three integer over/underflow bugs (lines 15, 16, and 31) where finding vulnerable transaction sequences for them is nontrivial. Understanding how the integer underflow at line 31 occurs is particularly tricky, although the existence of the bug seems apparent as there are no explicit guard statements (e.g. `require (totalSupply >= value)`) to prevent it from happening. For example, simply sending a transaction like `burnFrom (A, 1)` after deployment fails to trigger the bug, because all balances and allowances are initially zeros and therefore the transaction is aborted by the statement at line 27. To demonstrate the bug, we need to generate a transaction sequence of length at least 4, excluding an initial transaction (i.e. call to the constructor for deployment). For example, the bug can be triggered by the following scenario:

1. `mintToken(A,V1)` with `msg.sender` = `owner`
2. `approve(C,V2)` with `msg.sender` = `B`
3. `mintToken(B,V3)` with `msg.sender` = `owner`
4. `burnFrom(B,V4)` with `msg.sender` = `C`

where `A`, `B` and `C` are some account addresses, and `V1`–`V4` are crafted integer values, e.g., `V1=0x800...00`, `V2=10`, `V3=0x800...01`, and `V4=10`. What is tricky in this scenario is that, in order to trigger the integer underflow at line 31, a series of transactions must first conspire to exploit another bug in the contract (the integer overflow at line 16). Note that the first and third transactions cause `totalSupply` to overflow at line 16 and have an integer value at high risk of underflow; in the scenario above, `totalSupply` becomes `1` (`=0x800...00`+`0x800...01`). The second transaction is required for the last transaction to pass the guard statement at line 28. Finally, invoking `burnFrom` is able to cause the desired underflow bug at line 31.

SmarTest automatically generates the above transaction sequence and helps to diagnose and fix the root cause of the bug; to avoid the underflow at line 31, it is enough to insert a guard statement `require (totalSupply + amount >= totalSupply)` at the entry of the `mintToken` for preventing the overflow at line 16, without any modifications in the `burnFrom`. By contrast, existing tools do not help in this aspect. As mentioned earlier, most tools (e.g. [6, 24, 28, 36, 37]) are fundamentally improper for generating vulnerable transaction sequences. Two symbolic execution tools, Mythril [3] and Manticore [30], are ineffective too in this case; they fail to produce a sequence in 3 hours even when we hint that the maximum search depth is 4.

3 Approach

In this section, we describe our approach in detail. Section 3.1 describes the basic symbolic execution algorithm for discovering vulnerable transactions. Section 3.2 explains how to guide the symbolic execution with a language model. We use Figure 4 as a running example.

Language. We formalize our approach for a core subset of Solidity [4], which is defined by the following grammar [36]:

$$\begin{array}{lcl} c \in C & ::= & G^* \; F^*, \qquad F \; ::= \; f(x)\{S\} \\ a \in A & ::= & x := E \mid x[y] := E \mid \mathit{assume}(B) \mid \mathit{assert}^l(B) \\ s \in S & ::= & A \mid \mathit{if}\; B\; S_1\; S_2 \mid \mathit{while}\; B\; S \mid S_1 ; S_2 \end{array}$$

A contract C is a sequence of global variable declarations (G^*) followed by a sequence of function definitions (F^*). A function is comprised of a function name (f), a formal parameter (x), and a body statement (S). We denote the name of a constructor function by f_0. A statement S is an atomic statement A, an if-statement, a while-loop, or a sequence. An atomic statement A is an assignment to either a variable ($x := E$) or an array (or a `mapping` in Solidity) element ($x[y] := E$), an *assume* statement, or an *assert* statement. E and B are usual arithmetic and boolean expressions, respectively. We assume E evaluates to an unsigned 256-bit integer.

In our language, *assume* statements are used to model guard expressions (i.e. B in if-statements, while-loops, or `require` statements in Solidity) when generating paths in Section 3.1. On the other hand, *assert* statements do not affect program semantics; they express safety properties to be verified or refuted. We assume every safety condition that needs to be checked is expressed as an *assert* statement. Note that users of SmarTest do not need to write safety conditions for checking common security vulnerabilities (e.g., integer overflows), because assertions that express the safety conditions are automatically inserted in the preprocessing step of SmarTest. For example, when we want to check whether a contract contains integer overflow vulnerabilities, given an assignment $x := y + z$, we assume an assertion $\mathit{assert}(y + z >= y)$ is inserted right before the assignment. Custom safety conditions can be provided using `assert` statements in Solidity. We assume every *assert* statement is annotated with a unique label l, which serves as an identifier for each assertion. Let L be the set of all labels in a program. We assume all functions have `public` or `external` visibilities (i.e., callable from outside). We assume functions that cannot be called from outside (i.e., `internal` or `private` functions) are inlined at each call site. We also assume all variables have primitive types (e.g. `uint`) or mapping types (e.g. `mapping(address => uint)`). These assumptions are for presentation brevity; our implementation supports most of the features in Solidity.

Transaction Sequence. Given a function $f(x)\{S\} \in F$, we say (f, x, a) is a *function path*, where $a \in A^*$ is a sequence of atomic statements from the entry to the exit of the function (we assume the body S of the function is transformed into

```
contract Example {
  bool flag;
  uint x;

  constructor () public { }

  function setX (uint y) public returns {
    x = y;
  }

  function setFlag (bool b) public returns {
    flag = b;
  }

  function incX () public returns {
    if (flag) {
      assert (x+1>=x); //goal: disprove the assertion
      x = x+1;
    }
  }
}
```

Figure 4: A running example.

a set of atomic statement sequences [8]). Let P be the set of all function paths in a given contract $c \in C$. We define a *transaction* $t \in T$ to be a four-tuple:

$$t = (id, f, x, a)$$

which is a function path augmented with a transaction identifier id. We note that multiple transactions can be generated from a single function because the function may have multiple paths (e.g., the `incX` function in Figure 4 has two paths). We write t_0 for an initial transaction, i.e., a transaction whose second component (function name) is the name of a constructor (f_0). A *transaction sequence* $(t_0, t_1, \ldots, t_n) \in T^*$ is a series of transactions that start from an initial transaction. We say a transaction sequence $(t_0, t_1, \ldots, t_n)$ is vulnerable when the function called by the last transaction t_n contains an assertion (i.e., safety condition) that can be violated along the sequence.

Goal. In this paper, we tackle the problem of finding as many *vulnerable transaction sequences* as possible with concrete argument values for the parameters of involved transactions.

3.1 Basic Symbolic Execution

Algorithm 1 shows the overall symbolic execution algorithm. The input is a Solidity smart contract c, and the output is a report that shows a vulnerable transaction sequence disproving the safety condition of each assertion in c. The algorithm consists of a preparation step (lines 1–4) and a main analysis phase (lines 5–16).

To avoid generating function paths indefinitely, we first unroll all loops m times and inline each function into its call site up to n nested calls (line 1). From the resulting contract c, we collect function paths in c until up to o paths are gathered for each function in c (line 2). In the current implementation, we set m, n, and o to 2, 3, and 50, respectively. The algorithm initializes the workset W with initial transactions

Algorithm 1 Our Symbolic Execution Procedure for Finding Vulnerable Transaction Sequences

```
Input: A Solidity smart contract $c$
Output: A vulnerability report $R$
 1: Unroll loops and inline function calls in $c$
 2: $P \leftarrow$ The set of function paths in $c$
 3: $W \leftarrow \{(id, f_0, x, a) \mid (f_0, x, a) \in P,\ \text{new } id\}$
 4: $R \leftarrow \lambda l.\bot$
 5: repeat
 6:     $s \leftarrow \text{argmin}_{w \in W}\ \text{cost}(w)$                ▷ $s = t_0, t_1, \ldots, t_n$
 7:     $W \leftarrow W \setminus \{s\}$
 8:     $(State, \Pi) \leftarrow \text{GENERATEVC}(s)$
 9:     for each $(l, VC) \in \Pi$ do
10:         if $R(l) = \bot$ then                ▷ $l$ is not yet falsified
11:             if $\text{SAT}(VC)$ then $R \leftarrow [l \mapsto (s, \text{model}(VC))]$
12:     end for
13:     if $\text{SAT}(State)$ or Solver timeout then
14:         $W \leftarrow W \cup \{s \cdot (id, f, x, a) \mid (f, x, a) \in P,$
                                  $f \neq f_0,\ \text{new } id\}$
15:     end if
16: until $W = \emptyset$ or $\forall l.\ R(l) \neq \bot$ or timeout
17: return $R$
```

(line 3). During the algorithm, the workset W keeps candidate transaction sequences to be explored. At line 4, the algorithm also initializes the report $R : L \to T^* \times Model$, i.e., mapping from assertion labels to vulnerable transaction sequences with error-triggering input values (models), where $\lambda l.\bot$ (line 4) means that no vulnerable transaction sequences are found yet for any assertions.

The algorithm enters the loop at lines 5–16, which iteratively searches for vulnerable transaction sequences. The algorithm picks a candidate transaction sequence $s = t_0, t_1, \ldots, t_n$ with the least cost (line 6) and remove it from W (line 7), where t_0 is an initial transaction. At the moment, given a sequence s, we assume the cost function is defined as $\text{cost}(t_0, t_1, \ldots, t_n) = n$, which outputs the length of the transaction sequence. That is, the current cost function simply prioritizes short transaction sequences. This cost function will be replaced by the language model-guided cost function in Section 3.2.

After picking the candidate s, we perform symbolic execution over s to obtain a state condition ($State$) for s and verification conditions (VCs) (line 8), where Π is a set of pairs of an assertion label l and the VC associated with l. The VCs are conditions that must be checked to see whether s is a vulnerable sequence with respect to some assertions; the satisfiability of the VCs implies the existence of vulnerable transaction sequences. We will explain the VC generation procedure (GENERATEVC) shortly in Section 3.1.1. We investigate each of the VCs through the inner loop at lines 9–12. If a vulnerable transaction sequence with respect to an assertion annotated with l is already found (i.e., $R(l) \neq \bot$), we move on to the next VC (i.e., we do not attempt to disprove assertions whose safety conditions are already falsified by other transaction sequences). For assertions that are not disproved yet (i.e., $R(l) = \bot$, line 10), we check the satisfiability of the VC by invoking an off-the-shelf SMT solver (we use Z3 [13]). If satisfiable (i.e., s is a vulnerable transaction sequence), we update the report R by mapping l to s with a corresponding satisfying model (line 11).

Finally, if the state condition $State$ is satisfiable (i.e., s is a feasible transaction sequence in concrete execution) or a predetermined solver timeout expires (line 13), we generate a set of new transaction sequences by appending new transactions to the current sequence s, and add the set into the workset W (line 14). Otherwise (i.e., $State$ is unsatisfiable), we do not collect new transaction sequences, because further explorations of unsatisfiable states will not produce satisfiable VCs. The loop repeats until the workset becomes empty, vulnerable sequences for all assertions are found, or a given time limit is reached. At line 17, the symbolic execution procedure finally returns R, from which we can obtain a vulnerable transaction sequence (with concrete input values) for each potentially violated assertion.

3.1.1 VC Generation

We describe the GENERATEVC procedure for generating verification conditions, which symbolically executes a transaction sequence and derives a condition to be vulnerable. We first define symbolic execution for atomic statements and extend it to transactions and their sequences.

Let FOL be the set of the first-order formulas in the combined theory of fixed-sized bitvectors and arrays with extentionality. Let $\mathsf{sp} : A \to \mathsf{FOL} \times \wp(L \times \mathsf{FOL}) \to \mathsf{FOL} \times \wp(L \times \mathsf{FOL})$ be the strongest postcondition predicate transformer [8], which symbolically executes each atomic statement as follows:

$$\begin{aligned}
\mathsf{sp}(x := e)(\phi, \Pi) &= (\phi[x'/x] \land (x = e[x'/x])^{\circ}, \Pi) \\
\mathsf{sp}(x[y] := e)(\phi, \Pi) &= (\phi[x'/x] \land (x = x'\langle y \lhd e[x'/x]\rangle)^{\circ}, \Pi) \\
\mathsf{sp}(\mathit{assume}(e))(\phi, \Pi) &= (\phi \land e^{\bullet}, \Pi) \\
\mathsf{sp}(\mathit{assert}^{l}(e))(\phi, \Pi) &= (\phi, \{(l, \phi \land \neg e)\} \cup \Pi)
\end{aligned}$$

where $\phi[x'/x]$ and $e[x'/x]$ denote the formula ϕ and expression e, respectively, where x is replaced by x'. The definition is mostly standard; it transforms a precondition ϕ into a postcondition with respect to a given atomic statement, while accumulating Π (pairs of assertion labels and corresponding VCs). In the assignment cases ($x := e$ or $x[y] := e$), a primed variable (e.g., x') represents the previous state of an unprimed variable (e.g., x) before processing the assignments. We write $x'\langle y \lhd e\rangle$ for the array x' whose element at index y is replaced by e. In the assertion case, we collect a verification condition ($\phi \land \neg e$) by pairing it with the label l of the assertion to provide a potentially violating sequence per assertion (line 11 in Algorithm 1). Observe that a VC consists of two parts: a condition denoting a program state (ϕ) and a negation of a safety condition ($\neg e$). An unusual part in the syntax of FOL is that each atomic formula can be annotated with either a symbol $\circ$ or $\bullet$. We introduce these symbols to simplify constraints by differentiating equalities from assignments and *assume* statements (Section 3.1.2). These symbols will be removed before invoking SMT solvers.

Next, we define $\mathrm{T} : T \to \mathsf{FOL} \times \wp(L \times \mathsf{FOL}) \to \mathsf{FOL} \times \wp(L \times \mathsf{FOL})$, a symbolic executor for a transaction $t = (i, f, x, (a_1, \cdots, a_n))$:

$$\begin{aligned}
&\mathrm{T}(i, f, x, (a_1, \cdots, a_n))(\phi, \Pi) = \\
&\quad (\textsc{RenameL}(\phi', i), \{(l, \textsc{RenameL}(F, i) \mid (l, F) \in \Pi'\})
\end{aligned}$$

where $(\phi', \Pi') = \mathsf{sp}(a_n) \circ \cdots \circ \mathsf{sp}(a_1)(\phi \land x^e = x \land \varphi, \Pi)$. RENAMEL is a function for differentiating local variables with the same names in different transactions. More concretely, RENAMEL renames all free variables in a given formula, except for global variables, primed global variables, and variables that are already renamed using other transaction identifiers while processing previous transactions. For example, if $G = \{a\}$ (i.e., a is the only global variable in a contract), $\textsc{RenameL}(a' = 0 \land b = 1 \land c' = 2, j)$ outputs $(a' = 0 \land b_j = 0 \land c'_j = 1)$, where a' is not renamed since its original unprimed version is the global variable a. Observe that the procedure T proceeds in two steps. Firstly, given a precondition and label-VC pairs Π, we obtain the postcondition ϕ' and the possibly updated pairs Π', by symbolically executing $a_1, \cdots, a_n$ with sp. Secondly, we postprocess ϕ' and Π' with RENAMEL. Note that we have additional preconditions ($x^e = x$ and φ) in the first step. $x^e = x$ is a constraint for retrieving argument values for each transaction, where x^e is an entry-state variable for the formal input parameter x (i.e., the state of x at the entry of each transaction). φ is a conjunctive formula, which is a Solidity-specific constraint for obtaining useful arguments. For example, conjuncts of φ include $\mathit{msg.sender} \neq 0$ to avoid obtaining invalid values for transaction senders. We assume that each conjunct of φ is labelled with $\bullet$, to ensure that these constraints are not removed by our simplification technique (Section 3.1.2).

Finally, we define the procedure GENERATEVC that performs symbolic execution over a transaction sequence $t_0, t_1, \cdots, t_n$. $\textsc{GenerateVC}(t_0, t_1, \cdots, t_n)$ outputs (State, Π) where $(\mathit{State}, \Pi) = \mathrm{T}'(t_n) \circ \cdots \circ \mathrm{T}'(t_0)(\bigwedge_{g \in G} \mathsf{init}(g), \emptyset)$. A symbolic executor T' for a transaction t_i is defined as:

$$\mathrm{T}'(t_i)(\phi'', \Pi'') = \begin{cases} (\phi', \Pi') & \text{if } i = n \\ (\phi', \emptyset) & \text{otherwise} \end{cases}$$

where $(\phi', \Pi') = \mathrm{T}(t_i)(\phi'', \Pi'')$. Note that, given a transaction sequence, we collect VCs from the last transactions only (i.e., when $i = n$) and do not redundantly collect VCs from prior transactions, because further explorations of t_i ($i < n$) do not help to disprove safety conditions in t_i. $\emptyset$ means no VCs are collected yet. $\mathsf{init}(g)$ generates a constraint on a declaration of the global variable $g \in G$:

$$\mathsf{init}(g) = \begin{cases} x = \mathbf{0} & \text{if } g = x \\ \forall y.x[y] = \mathbf{0} & \text{if } g = x[y] \end{cases}$$

where $\mathbf{0}$ means a default value for each type of a variable or an element, e.g., $\mathbf{0}$ is *false* for `bool` types and 0 for `uint` types.

Example 1 *Consider the contract in Figure 4. Suppose we are given a transaction sequence* $t_0 \cdot t_1 \cdot t_2 \cdot t_3$ *where*

$$t_0 : (p, constructor, \bot, \bot), \qquad t_1 : (q, setX, y, x := y),$$
$$t_2 : (r, setFlag, b, flag := b),$$
$$t_3 : (s, incX, \bot, (assume(flag); assert^l(x+1 \geq x); x := x+1)).$$

Then, $\textsc{GenerateVC}(t_0 \cdot t_1 \cdot t_2 \cdot t_3) = (-, \{(l, F)\})$ *where F is a VC for disproving the safety condition at line 17 of Figure 4:*

$$x' = 0 \land flag' = false \land y^e_q = y_q \land x = y_q \land$$
$$b^e_r = b_r \land flag = b_r \land flag \land \neg(x+1 \geq x)$$

where we assume symbols ∘ and • are removed. Observe that the formal parameters (y and b) and the corresponding entry-state variables (y^e and b^e) are renamed as y_q, b_r, y^e_q, and b^e_r using the transaction identifiers respectively, because they are neither the global variables nor the primed global variables.

3.1.2 Constraint Simplification

Constraint solving is a major performance bottleneck in symbolic execution [7, 15, 21] and it was no exception in SmarTest. We devised two constraint simplification techniques to boost SMT solvers, which are particularly effective in the context of Algorithm 1. We apply these techniques right before line 11 and line 13 in Algorithm 1.

Property-focused Simplification. Firstly, we identify and remove parts of the VC that are unnecessary for generating vulnerable transaction sequences. More specifically, our technique aims to remove *redundant* constraints that include unnecessary variables. Unnecessary variables are variables whose values affect neither atomic formulas annotated with • (i.e., path conditions or Solidity-specific constraints, Section 3.1.1), nor the safety condition in the VC. For example, consider the following VC (we omit Solidity-specific constraints for brevity):

$$(x = y)^{\circ} \land (z = 10)^{\bullet} \land \neg(y+1 \geq y).$$

The VC may be generated from a sequence of atomic statements $x := y; assume(z = 10); assert^l(y+1 \geq y)$ where $(x = y)^{\circ} \land (z = 10)^{\bullet}$ is a state condition and $y+1 \geq y$ is a safety condition. Observe that the constraint $(x = y)^{\circ}$ can be removed as the path and safety conditions have no dependencies on the variable x defined in the statement $x := y$. Note that, without the symbol ∘, we would not be able to identify x as an unnecessary variable, because the information on the direction of value flow (i.e. from y to x, but not vice versa) is lost. Using this information, our technique simplifies the formula above into the following:

$$z = 10 \land \neg(y+1 \geq y)$$

where the redundant constraint $(x = y)^{\circ}$ is removed. Observe that we can find a satisfying model $[z \mapsto 10, y \mapsto 2^{256} - 1]$. We explain the detailed procedure in Appendix A.

Quantifier Elimination. Secondly, we transform constraints with quantifiers into quantifier-free constraints. We devised this technique because we observed that existing SMT solvers are often inefficient to handle formulas with universal quantifiers that are introduced by initializations of mapping-typed variables (Section 3.1.1). Our key insight for eliminating those universal quantifiers is that, in many cases it is enough to replace quantified variables by indices for certain elements that appear in a given formula. For example, consider a VC

$$\forall i.x'[i] = 0 \land x = x'\langle y \lhd 10 \rangle \land \neg(x[y] < 10)$$

where x is a mapping type (`mapping(address=>uint)`) variable, and y is an address type variable where addresses are 160-bit expressions in Solidity. Our technique transforms the formula into its quantifier-free version

$$x'[y] = 0 \land x = x'\langle y \lhd 10 \rangle \land \neg(x[y] < 10)$$

by initializing the element of x' at index y only, because the y is the only index variable that is used to access the element of the mapping x' and its unprimed version x (which shares the element of x'). We explain the detailed procedure in Appendix B.

3.2 Symbolic Execution with Language Model

Now we present the key technical contribution of this paper, i.e., guiding symbolic execution with a language model for effectively finding vulnerable transaction sequences.

Background on Language Model. We provide a necessary background on language model based on [23]. A language model assigns a probability to each sequence $w_1, \cdots, w_n$ of words. The probability is denoted $P(w_1, \cdots, w_n)$ and quantifies how likely or natural the sentence is. $P(w_1, \cdots, w_n)$ can be computed using the chain rule of probability:

$$P(w_1, \cdots, w_n) = \prod_{i=1}^{n} P(w_i \mid w_1 \cdots w_{i-1}).$$

However, using the chain rule may not generalize to unseen data due to the data sparsity problem. While there exist numerous language models to deal with the data sparsity, we use the n-gram model that is found to be simple yet effective for our purpose. The n-gram model uses the last $(n-1)$ words as context when predicting the last word. For example, for 3-gram model, $P(w_1, \cdots, w_n)$ is estimated as follows:

$$P(w_1, \cdots, w_n) \approx \prod_{i=1}^{n} P(w_i \mid w_{i-2} w_{i-1})$$

where each probability can be computed by maximum likelihood estimation, i.e., normalizing 3-gram counts by previous 2-gram counts from the training corpus:

$$P(w_i \mid w_{i-2}w_{i-1}) = \frac{C(w_{i-2}w_{i-1}w_i)}{C(w_{i-2}w_{i-1})}.$$

For illustration, we will explain our approach with the 3-gram model in the rest of this section.

Role of Language Model. We leverage a language model to compute probabilities for predicting how likely a given (partial) transaction sequence will become vulnerable in the future, which we call *vulnerable probabilities* of transaction sequences. With vulnerable probabilities, we can steer symbolic execution towards effectively finding vulnerable transaction sequences.

The rest of this section is organized as follows: learning a language model from collected vulnerable transaction sequences (Section 3.2.1), and guiding symbolic execution with a learned language model (Section 3.2.2).

3.2.1 Learning a Language Model

Training an n-gram language model is essentially to collect n-gram counts (e.g., $C(w_iw_{i+1}w_{i+2})$) from word sequences (e.g., $w_1 \cdots w_n$), where the counting information will be used to compute vulnerable probabilities (Section 3.2.2). That is, our goal in the training phase is to construct a training corpus (i.e., a multiset of word sequences) Y and then collect n-gram counts from it.

Collecting Vulnerable Transaction Sequences. Firstly, we collect a multiset of vulnerable transaction sequences $\{T_1, \cdots, T_m\}$ (where $T_i \in T^*$) by running our basic symbolic execution (Section 3.1) on training data with sufficient time budget (30 minutes per contract in our experiments, see Section 5).

Note that, if we treat a transaction itself without any abstractions as a word, a learned language model would not generalize to unseen data, because there are possibly many syntactic variations in real-world smart contracts, though overall structures of the code are similar. For example, the two-dimensional mapping variables `allowance` in Figure 2 and Figure 3 sometimes appear with different names (e.g., `allowed`) in other contracts.

Abstracting Transaction Sequences. To make a learned language model better generalize to unseen contracts, we represent a transaction as an abstract form. Our key idea for representing transactions as words is to use type information. We observed that type information plays an important role for characterizing behaviors of functions in smart contracts. More specifically, functionalities of Solidity smart contracts are often implemented in modular ways and, as a result, each function involves only certain types of variables out of the whole set of variables in a contract. For example, for a function involving a global variable of a type `mapping(address => mapping(address => uint))`, a Solidity developer may be able to come up with `approve` function that frequently appears in ERC20-based token contracts (e.g., Figure 2 and Figure 3).

Based on the observation, we obtain final training corpus Y by transforming each transaction sequence $T_i = t_i^0 \cdots t_i^n$ into a corresponding word sequence:

$$Y = \{\langle s\rangle\langle s\rangle\alpha_\tau(t_i^0)\cdots\alpha_\tau(t_i^n)\langle e\rangle\langle e\rangle \in \widehat{T}^* \mid T_i = t_i^0 \cdots t_i^n, i \in [1,m]\}.$$

$\tau : Type \to \mathbb{N}$ is a type frequency table that maps each type to the number of its occurrences from the collected transaction sequences $\{T_1, \cdots, T_m\}$. Specifically, we obtain τ by counting type frequencies for global variables that are defined via assignments or used in *assume* statements within each transaction in $\{T_1, \cdots, T_m\}$. Using τ, a word map $\alpha_\tau : T \to \widehat{T}$ abstracts a transaction to a *word* (an abstract form of a transaction), which is defined as follows:

$$\alpha_\tau(t) = \begin{cases} \textbf{if } t = (-, f_0, -, -) \textbf{ then } \langle i\rangle \\ \textbf{else} \langle D_\tau^1(t), \cdots, D_\tau^k(t), U_\tau^1(t), \cdots, U_\tau^k(t), P(t), E(t), X(t)\rangle. \end{cases}$$

Note that the set of words $\widehat{T} = \{\langle s\rangle, \langle e\rangle, \langle i\rangle\} \cup \{0,1\}^{2k+3}$. That is, a word $w \in \widehat{T}$ is either a pseudo-start word $\langle s\rangle$, a pseudo-end symbol $\langle e\rangle$, a constructor word $\langle i\rangle$ for abstracting initial transactions t_0, or a boolean vector of $2k+3$ dimension. Further note that we consider only top k-th ranked types from τ for generalization (i.e., we use τ as a criterion for identifying types that are important for abstract representations of transactions). Let a be a sequence of atomic statements of a transaction t (i.e., $t = (-, -, -, a)$). D_τ^i $(1 \le i \le k)$ is a predicate (1 for *true*, 0 for *false*) that checks whether a global variable, having a top i-th ranked type in τ, is defined via assignments in a of t. U_τ^i $(1 \le i \le k)$ is a predicate that checks whether a global variable, having a top i-th ranked type in τ, is used in *assume* statements in a. P, E, and X are additional, Solidity-specific feature predicates. P checks whether a function f is annotated with `payable` keyword. E checks whether a built-in function that sends Ethers (e.g., `transfer`) exists in a. X checks whether a built-in function that destructs a contract (`selfdestruct`, `suicide`) exists in a. Following the convention of 3-gram models [23], we append pseudo words $\langle s\rangle \cdot \langle s\rangle$ at the beginning of each word sequence and append $\langle e\rangle \cdot \langle e\rangle$ at the end of each word sequence.

Example 2 *Assume $k = 2$ and $\tau = [$`uint` $\mapsto 10,$ `bool` $\mapsto 3,$ `uint8` $\mapsto 1]$. Then, the transaction t_3 in Example 1 is represented as $\langle 1,0,0,1,0,0,0\rangle$, because* `uint` *type global variable is defined (thus the first component is set to 1),* `bool` *type global variable is used in assume (thus the fourth component is set to 1), and the function* `incX` *is not annotated with* `payable` *keyword and does not have statements that send Ethers or destruct the contract.*

Discussion. Let us justify our design choices on the transaction representation in more detail. For initial transactions $t_0 = (-, f_0, -, -)$, we uniformly abstract them into the special word $\langle i \rangle$ for generalization; for virtually all smart contracts, the main job of the constructor is to initialize global variables, rather than performing other specific functionalities. Note that, for D_τ^is and U_τ^is, we focus on types of global variables and ignore types of local variables, because Ethereum smart contracts are stateful and global states are affected by global variables only. We consider D_τ^is, P, E, and X in the representation, because they are important clues for understanding semantic behaviors in Solidity contracts. As an example for D_τ^is, consider a `transfer` function that is one of the core functions in ERC20 token contracts; it is common for a global variable of type `mapping (address=>uint)` to be defined, because the `transfer` function is in charge of transferring tokens from one's balance to another. We also consider U_τ^is in the representation, because they are important clues for inferring which transaction may have been called before. For example, to disprove the assertion at line 17 of `incX` in Figure 4, we first should set `flag` to *true* by invoking `setFlag`.

3.2.2 Using a Language Model

Let $V \subseteq \widehat{T}$ be a vocabulary, a set of known words from training sentences Y (Section 3.2.1), i.e., $V = \{w_i \mid w_1 \cdots w_m \in Y, i \in [1,m]\}$. Note that we can now compute vulnerable probabilities using n-gram counts from Y. Guiding symbolic execution with a language model is a two-step process.

Firstly, for a given transaction sequence $t_0 \cdots t_n$, we translate the transaction sequence into a word sequence $\langle s \rangle \cdot \langle s \rangle \cdot w_0 \cdots w_n$ where $w_i = \alpha'_\tau(t_i)$. Here, $\alpha'_\tau : T \to \widehat{T}$ is an extended word map for handling unknown words:

$$\alpha'_\tau(t) = \begin{cases} \alpha_\tau(t) & \text{if } \alpha_\tau(t) \in V \\ \text{argmax}_{w \in V}\, \mathsf{similarity}(\alpha_\tau(t), w) & \text{if } \alpha_\tau(t) \notin V \end{cases}$$

where $\mathsf{similarity}(w_1, w_2)$ is a function that heuristically computes the similarity between words:

$$\mathsf{similarity}(\langle v_1, \cdots, v_{2k+3} \rangle, \langle v'_1, \cdots, v'_{2k+3} \rangle) = \sum_{i=1}^{2k+3} N_i \times v_i \times v'_i.$$

That is, if we encounter an out-of-vocabulary word that is not obtained in the training phase (i.e., $\alpha_\tau(t) \notin V$), we transform it into the most similar word in the dictionary. For integer constants $N_1, \cdots, N_{2k+3}$ that represent weights for each feature vector, we set them to be $N_1, \cdots, N_{2k} < N_{2k+1}, N_{2k+2}, N_{2k+3}$, i.e., we give the highest scores for the Solidity-specific features. Moreover, note that we do not append $\langle e \rangle$s for each transaction sequence when computing vulnerable probabilities; our purpose is to estimate whether further explorations of a given transaction sequence would be beneficial for finding vulnerable transaction sequences, not to evaluate whether the given sequence itself is a vulnerable transaction sequence.

Next, we guide symbolic execution by redefining the cost function at line 6 of Algorithm 1 as follows:

$$\mathsf{cost}(t_0, \cdots, t_n) = -\prod_{i=0}^{n} P(w_i \mid w_{i-2} w_{i-1})$$

where $w_{-2} = w_{-1} = \langle s \rangle$ and $w_j = \alpha'_\tau(t_j)$ if $j \in [0,n]$. Note that we compute negative probabilities, because our algorithm picks a candidate with the least cost (Algorithm 1). Moreover, to make our language model generalize to unknown contexts that may appear in unseen contracts, we use a smoothing method called simple linear interpolation [20, 23], which mixes 1-gram, 2-gram, and 3-gram all together:

$$\begin{aligned} P(w_i \mid w_{i-2} w_{i-1}) \quad = \quad & \lambda_1 P_{\text{add}-\text{k}}(w_i \mid w_{i-2} w_{i-1}) + \\ & \lambda_2 P_{\text{add}-\text{k}}(w_i \mid w_{i-1}) + \\ & \lambda_3 P(w_i) \end{aligned}$$

such that $\lambda_1 + \lambda_2 + \lambda_3 = 1$ (condition for ensuring $\sum_{w_j \in V} P(w_j \mid w_{i-2} w_{i-1}) = 1$). Observe that, to compute 3-gram and 2-gram probabilities, we use add-k smoothing, e.g., for 3-gram, $P_{\text{add}-\text{k}}(w_i \mid w_{i-2} w_{i-1}) = \frac{C(w_{i-2} w_{i-1} w_i) + k}{C(w_{i-2} w_{i-1}) + k|V|}$ for some real number k (where $0 < k < 1$) to avoid zero counts in denominators. For 1-gram, we compute it by unsmoothed maximum likelihood estimation (i.e., $P(w_i) = \frac{C(w_i)}{\sum_{w_j \in V} C(w_j)}$), because numerator and denominator are not zeros (i.e., $\forall w \in V. C(w) > 0$). We remark that, though there exists a method called EM algorithm [20, 23] for obtaining locally optimal λ_is, we simply set $\lambda_1 = 0.9$, $\lambda_2 = 0.08$ and $\lambda_3 = 0.02$, which worked fairly well in our case.

4 Implementation

We implemented SMARTEST in OCaml on top of VERISMART [36], an open-sourced verifier for Solidity contracts. Specifically, we reused the frontend of VERISMART and its VC generator for atomic statements, but newly implemented our symbolic executor for transaction sequences (Section 3.1), constraint solving optimization (Section 3.1.2), and symbolic execution with a language model (Section 3.2).

To be practical as much as possible, our VC generator is implemented in a more sophisticated way than the one defined in Section 3.1. In particular, our VC generator takes into account statements that follow assertions. For example, consider the statements `uint x = n - 12; require (n==10); ...` where `n` is a formal input parameter. In this case, in order for the input value `n` to trigger the underflow for `n-12`, SMARTEST produces a value of `10`; while any integer values from `0` to `11` can trigger the underflow, the most desirable value would be `10`, since the effect by the underflow can persist after the transaction. Other extensions include the following.

Vulnerability Checker. Currently, SMARTEST supports the detection of six types of security-critical vulnerabilities: integer over/underflow, assertion violation, division-by-zero, ERC20 standard violation, Ether-leaking vulnerability (e.g., unauthorized access to `transfer`), and suicidal vulnerability (e.g., unauthorized access to `selfdestruct`). For the first three types of vulnerabilities, we reused the implementation of VERISMART. We provide detailed explanations on how we detect the rest three types of vulnerabilities (which are currently not supported by VERISMART) in Appendix C.

Concrete Validator. We implemented our concrete validator in Python using the Truffle testing framework.[4] We use the validator to confirm true positives (i.e., vulnerable transaction sequences generated by SMARTEST can violate corresponding safety conditions in concrete execution), thereby reducing the burden of manual effort for validating analysis results. Given a set of vulnerable transaction sequences obtained by running symbolic execution (Section 3) on a contract, our validator examines the analysis results as follows. First, on a testnet, we deploy contracts with assertions that are automatically generated for each safety condition deemed violated in the analysis phase. Next, we check safety conditions in assertions are falsified or relevant logging messages are emitted. For leaking and suicidal vulnerabilities, in addition to checking violations of safety conditions, to further increase the confidence in our analysis results, we check a positive amount of Ethers (produced by the symbolic execution) is indeed transferred to *untrusted* accounts (Appendix C) and an analyzed contract is actually deactivated, respectively. In our experiments (Section 5), we sampled validation results and manually confirmed that our validator works as intended.

Function Call Analysis. Although we described our approach for a small subset of Solidity, our implementation supports most of the features in Solidity, including internal function calls, inheritance, and structures. However, SMARTEST currently does not precisely handle external function calls (i.e., calling functions defined in other contracts). For example, given a statement `o.foo()` where `o` is a contract object, we produce the constraint *false* to reduce false positives (i.e., generated vulnerable sequences do not violate corresponding safety conditions in concrete execution). Also, given a `call` statement (e.g., `rcv.call.value(amount)()`), we consider Ether-transfer to detect leaking vulnerabilities, but do not consider behaviors of fallback functions of the Ether receiver (e.g., `rcv`).

5 Evaluation

In this section, we evaluate our approach to answer the following research questions.

- How effectively does SMARTEST find vulnerable transaction sequences? How does it compare to existing tools? (Section 5.1)
- Is using language models essential for performance? How does SMARTEST compare to the basic symbolic execution without language models? (Section 5.2)
- What are the insights we can get from the learned language models? (Section 5.3)
- Can SMARTEST find zero-day bugs from smart contracts in the wild? (Section 5.4)

[4]https://www.trufflesuite.com/

5.1 Effectiveness of SMARTEST

We evaluate the vulnerability-finding ability of SMARTEST in comparison with five publicly available tools: MYTHRIL [3], MANTICORE [30], MAIAN [31], TEETHER [25], and ILF [19]. They are well-known and recently-developed analyzers that can generate vulnerable transaction sequences. The first four are symbolic executors and the last one is a fuzzer. We compare these tools against five security-critical vulnerabilities: integer over/underflow, assertion violation, division-by-zero, Ether-leaking, and suicidal. The first three types of vulnerabilities are supported by MYTHRIL and MANTICORE. The leaking (resp., suicidal) vulnerabilities are supported by all tools (resp., all but TEETHER). We additionally demonstrate the effectiveness of SMARTEST on finding ERC20 standard violations, which is not supported by the five.

Benchmark Setup. One important issue when using machine learning approaches is about how to obtain sufficient amounts of useful data (i.e., sufficiently many vulnerable smart contracts). For vulnerabilities related to arithmetic properties (integer over/underflow, assertion violation, division-by-zero, ERC20 violation), we used smart contracts with known arithmetic vulnerabilities (e.g., integer overflows) reported in CVE. Out of the 487 smart contracts with arithmetic vulnerabilities, we used 443 contracts after we deduplicate contracts whose names of the root contracts and the code are exactly equivalent to previously collected ones. On average, the deduplicated contracts consist of 229 lines. We note that, although CVE dataset is mostly comprised of smart contracts with known integer over/underflow vulnerabilities, we additionally targeted three more kinds of vulnerabilities related to arithmetic properties, in order to compare the tools from more diverse perspectives.

We note that contracts in CVE dataset have several typical CVE-reported vulnerability patterns with some contract-specific variations (e.g., implementations of vulnerable functions, contract sizes), which thus can be useful enough to compare the effectiveness of tools. The vulnerability patterns include: over/underflows due to logical flaws in guard expressions (e.g., CVE-2018-12025), missing overflow protection statements in `mintToken` functions (e.g., CVE-2018-13085), and missing guard statements for preventing overflows in calculating the amount of tokens to be sent when sending tokens to multiple accounts (e.g., CVE-2018-10299—well-known for batchOverflow). The most of the benchmarks are ERC20 token contracts, reflecting the prevalence of them in the Ethereum blockchain.

For Ether-leaking and suicidal vulnerabilities, we used 104 smart contracts (90 contracts with leaking vulnerabilities and 53 contracts with suicidal vulnerabilities, Table 3) labelled with vulnerable program points (explained shortly). Out of

104, 50 contracts came from a publicly available vulnerability database, called SmartBugs [14], for Solidity smart contracts. Specifically, 8 out of 50 are mostly small test contracts manually collected by the authors of [14] from public repositories (e.g., SWC registry). The rest 42 contracts are the ones found by MAIAN [31]. More concretely, the authors of [14] ran MAIAN on deployed contracts (> 10,000), where MAIAN flagged 44 contracts in total (for Ether-leaking and suicidal vulnerabilities) out of them; we excluded 2 out of 44 as 2 were obtained by running against non-main contracts. We assumed contracts flagged by MAIAN have real vulnerabilities, since MAIAN internally performs concrete validation to confirm true positives [31]. One typical vulnerability pattern in contracts found by MAIAN is an improper access control due to a mistakenly named constructor (e.g., Pattern 1 in Section 5.4). For more extensive evaluation, we additionally collected the rest 54 (=104-50) contracts by manually injecting realistic vulnerabilities into 21 randomly selected real-world contracts deployed on blockchain. On average, the contracts (without duplicated contracts) in Leaking-Suicidal datatset consist of 335 lines.

We explain our constructed benchmarks in more detail. To mimic real vulnerabilities as possible, we tried not to excessively alter original code. Specifically, we devised and applied 3 mutation patterns in Appendix D, where mutation operations are negation or removal (rather than insertion of code) for preferring small changes. We also considered variations in program sizes; the 21 seed contracts consist of 399 lines on average, including 7 relative large contracts (> 500 lines). Similar to those in CVE dataset, these contracts are mostly token contracts, including a few other types of contracts (e.g., game).

Ground truths for Ether-leaking and suicidal vulnerabilities were manually constructed by the authors of [14] (ones that were provided for the 8 test contracts) and us (the rest, including ones unexpectedly found by tools used in our experiments). Although our ground truths for vulnerabilities may not be exhaustive despite our significant effort, we believe they will be useful for evaluating other analysis tools as well.

Although we tried hard in preparing benchmarks for objective evaluation, the benchmarks may not be perfect; unfortunately, however, we are currently unaware of other proper public datasets with confirmed vulnerabilities. We discuss the limitation of our benchmarks in Section 5.5.

Tool Setup. We obtained the latest versions (as of August, 2020) of each tool from public docker images provided by the developers of these tools (MYTHRIL, MANTICORE, ILF) and the public GitHub repository (TEETHER). For MAIAN, we were unable to run MAIAN obtained from its public repository due to library dependency issues. Instead, we used the docker image [5] provided by the authors of [14], where they managed the dependency issues for running MAIAN.

[5]https://hub.docker.com/r/smartbugs/maian/tags

For a fair comparison, we deactivated checker options that are irrelevant to vulnerabilities targeted in each experiment, when related options were available (MYTHRIL, MANTICORE). We ran MAIAN separately for each type of vulnerabilities, since it analyzes only one type of vulnerability for each run. For symbolic executors, we provided an option to explore transaction sequences up to length 4 when available (all but TEETHER), to avoid potential disadvantages of each tool in terms of the number of found vulnerable transaction sequences. Note that, for SMARTEST, we did not give such a hint on the transaction depth (Algorithm 1). For MANTICORE, we provided options for running it within the capacity of our machine, since MANTICORE creates multiple subprocesses per tool invocation by default. For all symbolic executors, we set the analysis timeout to 30 minutes per contract when timeout option is available (all but MAIAN and TEETHER), and we set the external timeout to 35 minutes per contract to ensure the termination. We ran ILF with 100K transactions until it terminates, which spent 37 minutes on average per contract. For additional inputs (other than contracts) required by TEETHER (e.g., the address of the attacker) and ILF (constructor argument values), we simply provided random values. We left remaining options as default for the four tools. For SMARTEST, we set timeout to 1 minute per Z3 request.

For CVE dataset, we ran each tool with 40 threads on Ubuntu machine with AMD Ryzen Threadripper 3970X CPU (3.7 GHz) (32 cores and 64 threads in total) and 62GB of memory. For the second dataset with leaking and suicidal vulnerabilities, we ran each tool with 26 threads on Ubuntu machine with two Intel(R) Xeon(R) E5-2630 v3 (2.40GHz) CPUs (16 cores and 32 threads in total) and 188GB of memory. As exceptions, we ran MAIAN and ILF with 2 and 3 threads respectively; we observed MAIAN produced runtime exceptions with 26 threads and ILF showed substantial CPU usage (e.g., > 500 %) per tool invocation.

Results. On each of the two datasets, we performed 4-fold cross validation, a methodology for evaluating the generalizability of models; we randomly divided each dataset into 4 folds with equal or similar sizes, ran each fold with our baseline symbolic execution (Section 3.1), and tested on each fold using a model learned from training sequences that were obtained from the rest three folds (i.e., each fold is used once as testing data and three times as training data). For each tool, following [19], we report numbers whose results on every fold is summed.

Table 1 provides the results on 443 contracts from CVE for each tool. The column #G shows the number of vulnerable transaction sequences found by each tool for each vulnerability kind and for each transaction depth. The column #V means the number of transaction sequences that are automatically confirmed by our validator; for CVE dataset, we also validated the outputs of MYTHRIL and MANTICORE using our validator. The results show that SMARTEST outperformed MYTHRIL and MANTICORE in terms of finding vulnerable

Table 1: Test results on 443 contracts with 4-fold cross validation. #G: the number of vulnerable transaction sequences generated by each tool. #V: the number of vulnerable transaction sequences confirmed by the validator. Each instance is unique, i.e., each of the instances in each contract is different in at least one of the following three things: lines, safety conditions, and vulnerability types.

Vuln. Kind	Tx. Depth	SmarTest		Mythril		Manticore	
		#G	#V	#G	#V	#G	#V
Integer Over/ Underflow	Total	2110	1982	594	460	4	2
	0	144	118	8	5	0	0
	1	890	862	442	354	3	1
	2	782	731	143	100	1	1
	3	287	264	1	1	0	0
	≥ 4	7	7	0	0	0	0
Division by Zero	Total	219	203	74	73	2	1
	0	0	0	0	0	0	0
	1	180	171	60	59	2	1
	2	38	31	14	14	0	0
	3	1	1	0	0	0	0
	≥ 4	0	0	0	0	0	0
Assertion Violation	Total	80	77	32	25	6	3
	0	0	0	0	0	0	0
	1	45	44	23	17	6	3
	2	31	30	8	7	0	0
	3	4	3	1	1	0	0
	≥ 4	0	0	0	0	0	0
ERC20 Standard Violation	Total	683	654	N/A		N/A	
	0	0	0				
	1	28	28				
	2	621	592				
	3	32	32				
	≥ 4	2	2				

transaction sequences. For example, for integer over/underflow vulnerabilities, SmarTest found 1,982 validated vulnerable transaction sequences. By contrast, Mythril and Manticore found 594 and 4 at most, respectively. We observe that SmarTest is particularly more effective in finding lengthy vulnerable transaction sequences (e.g. depth 3).

To evaluate the tools in a more security relevant aspect, we also compared three tools in terms of finding known CVE vulnerabilities related to integer over/underflows. We randomly sampled 300 out of 443 contracts and manually labelled vulnerable locations described in each CVE report. We found that 58 CVE reports are not valid (e.g., vulnerable functions reported in CVE cannot be invoked in designated main contracts, vulnerable functions reported do not exist in source code, determined to be incorrect [36]), or have integer overflow vulnerability patterns appeared in other CVE reports but the reports themselves are not directly related to overflows or the other types of vulnerabilities targeted in our experiment (e.g., CVE-2018-12078). Table 2 shows that SmarTest outperforms Mythril and Manticore in this aspect as well. SmarTest found 93.0% (225/242) of the known vulnerabilities in total; using our concrete validator, we checked that SmarTest successfully generated validated

Table 2: Evaluation on labelled 242 CVE reports out of randomly sampled 300 CVE reports. #G: the number of found CVE vulnerabilities (possibly spanning multiple lines per vulnerability). #V: the number of CVE vulnerabilities confirmed by the validator.

Sampled CVE	Labelled CVE	SmarTest		Mythril		Manticore	
		#G	#V	#G	#V	#G	#V
300	242	225	219	90	85	0	0

vulnerable sequences for 90.5% (219/242). On the other hand, Mythril and Manticore found 37.2% (90/242) and 0 of the known vulnerabilities in total, respectively. We note that the findings of SmarTest in Table 2 strictly include those of Mythril and Manticore. We also note that Mythril and Manticore produced analysis failures on 3 and 274 contracts, respectively.

On the leaking and suicidal contracts, SmarTest found more vulnerabilities compared to the five tools (Table 3). For a fair comparison as possible, we compare the six tools in three levels (contract, function, and lines), because we observed Maian and teEther immediately terminate once they found one vulnerability in each contract (i.e., they do not try to exhaustively find all vulnerable locations) and ILF reports vulnerable function names without line-level information. At contract level, SmarTest detected 90.0% (81/90) and 96.2% (51/53) of leaking and suicidal contracts with validated transaction sequences, whereas ILF (the best among the five tools) detected 83.3% (75/90) and 94.3% (50/53). SmarTest is consistently more effective than the five tools in both function- and line-levels. We observed that existing tools were less effective in finding leaking vulnerabilities, because it typically requires longer transactions than finding suicidal vulnerabilities, requiring steps for designating malicious Ether-receivers. We also observed interesting false negative cases for ILF. While ILF was effective in most cases, it failed to detect vulnerabilities when relatively tricky arguments are necessary for passing by guard statements. For example, ILF failed to detect the suicidal vulnerability in the following code snippet:

```
1  function kill (uint code) public /* onlyOwner */ {
2    require (code == 1234567890);
3    selfdestruct(owner);} // suicidal vulnerability
```

where we injected a vulnerability by removing the `onlyOwner` modifier (i.e., anyone can kill the contract). In the snippet, the developer's intention at line 2 was to prevent accidental invocation of this function. Two symbolic executors SmarTest and Maian found this vulnerability. We also note that SmarTest reported four warnings not in our ground truths, which are false positives (virtually safe though predefined safety conditions can be violated, excluded in Table 3) due to current imprecise modeling of leaking vulnerabilities. For Maian, we excluded one finding from Table 3, where it did not properly report a vulnerable function (e.g., the hash of the reported function did not match with any functions in the contract).

Table 3: Results on 104 contracts (90 with leaking and 53 with suicidal vulnerabilities). #G: the number of vulnerable transaction sequences. #V: the number of validated vulnerable transaction sequences; MAIAN provides its own validated results in concrete execution and we report them, and we deem #G = #V for ILF because ILF performs dynamic analyses. #Fail: the number of contracts on which each tool produced some failures without any partial results. #TO: the number of contracts on which each tool encountered timeout; we considered partial results when available (MANTICORE). n/a: relevant information is not available from results obtained by each tool, or tools immediately terminate once one vulnerability is found in a contract.

Tools	**Leaking (Total: 90 contracts)**								**Suicidal (Total: 53 contracts)**							
	Contract		Function		Line		#Fail	#TO	Contract		Function		Line		#Fail	#TO
	#G	#V	#G	#V	#G	#V			#G	#V	#G	#V	#G	#V		
SMARTEST	82	81	112	111	115	111	0	0	51	51	51	51	51	51	0	0
ILF	75	75	101	101	n/a	n/a	4	-	50	50	50	50	n/a	n/a	1	-
MAIAN	65	58	n/a	n/a	n/a	n/a	7	0	43	43	n/a	n/a	n/a	n/a	7	0
TEETHER	37	n/a	n/a	n/a	n/a	n/a	7	29	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
MYTHRIL	7	n/a	8	n/a	8	n/a	0	0	19	n/a	19	n/a	19	n/a	0	0
MANTICORE	9	n/a	9	n/a	9	n/a	65	9	3	n/a	3	n/a	3	n/a	46	4

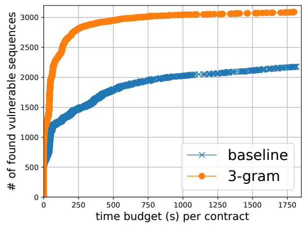
Analysis Cost. The runtime costs of each tool for obtaining the results on CVE dataset (Table 1 and 2) are: SMARTEST (6h 7m), MYTHRIL (6h 5m), and MANTICORE (4h 35m). The costs of each tool for obtaining the results on the second dataset (Table 3) are: SMARTEST (2h 4m), ILF (22h 49m), MAIAN (2h 28m), TEETHER (2h 20m), MYTHRIL (2h 5m), and MANTICORE (2h 21m).

Learning Cost. On average, the training time on CVE dataset can be computed as about 4.5 hours; the total time for running 4 folds with basic symbolic execution (Section 3.1) took about 6 hours (6h 8m) and 3 folds are used as training data for obtaining n-gram counts (i.e., 6 hours * 3/4). Similarly, the average learning time on Leaking-Suicidal dataset can be calculated as 1.5 hours, where the total running time on all four folds with basic symbolic execution is about 2 hours (2h 4m). Note that, given n-gram counts, computing vulnerable probabilities (Section 3.2) is done on demand during symbolic execution and thus requires no additional training time.

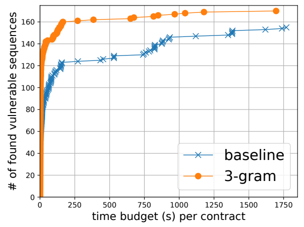
5.2 Effectiveness of Using Language Model

Figure 5 shows the performance of SMARTEST with and without language models. In Figure 5(a) (resp., (b)), the meaning of a point at (x,y) is as follows: from the 443 (resp., 104) contracts, y vulnerable transaction sequences were found in total when each contract was analyzed with the testing budget of x seconds. The two figures show that the learned language models greatly help to find more vulnerable transaction sequences in a short time. For example, on CVE dataset, while the basic symbolic execution took 1,817 seconds to find 2,178 vulnerable transaction sequences, our language model-guided symbolic execution took 68 seconds to find the same number of vulnerable transaction sequences.

Discussion. SMARTEST can be effective when it is trained and tested on contracts with vulnerabilities whose patterns of vulnerable transaction sequences are similar. However, SMARTEST may not be effective when trained and tested on



(a) CVE dataset



(b) Leaking-Suicidal dataset

Figure 5: SMARTEST with vs. without language model.

vulnerabilities whose typical patterns of vulnerable sequences are substantially different. We discuss our limitation with following experiments: training on CVE dataset for four types of vulnerabilities in Table 1 and testing on Leaking–Suicidal dataset for two types of vulnerabilities in Table 3, and vice versa. For each experiment, we observed a language model trained on one dataset degrades the performance of our basic symbolic execution when tested on the other dataset: 2,084 vulnerable transaction sequences (vs. 2,178) for the first experiment (trained on Leaking-Suicidal dataset and tested on CVE dataset), 149 sequences (vs. 155) for the second experiment (trained on CVE dataset and tested on Leaking-Suicidal dataset). One possible reason for these results may be that, typical patterns of vulnerable transaction sequences that appear in each dataset are rather different (e.g., Section 5.3). We believe generalizing to vulnerabilities with different sequence patterns is challenging and further research is needed for it.

5.3 Learned Insight

We present case studies that can help to understand how our learned language models improve the speed of symbolic execution. We have inspected learned conditional probabilities that are commonly high ranked in each model, where we considered top-6 types in each training phase (i.e., $k = 6$, see Section 3.2.1). For CVE dataset, we observed that prioritizing transactions without proper arithmetic guard statements is important for quickly finding arithmetic vulnerabilities. For example, consider the conditional probability below:

$$P(\langle 110000001000000\rangle \mid \langle s\rangle \cdot \langle i\rangle)$$

where `mapping(address => uint)` and `uint` are the top two variable types. We note that one possible implementation corresponding to $\langle 110000001000000\rangle$ is `mintToken` function (e.g., Figure 3), where 1st and 2nd elements are set to 1 (i.e., variables with the top two types may be defined) and 7th and 8th elements are set to 0 (i.e., corresponding guard statements do not exist). For Leaking-Suicidal dataset, we observed that finding transactions involved with unprotected ownership is critical for finding those two types of vulnerabilities. One example is the following conditional probability:

$$P(\langle 000000100000010\rangle \mid \langle i\rangle \cdot \langle 110000000000000\rangle)$$

where `address` is the top ranked variable type. $\langle 110000000000000\rangle$ may indicate a transaction that enables to change contract's owners without checking access privileges (the 1st and 7th elements are set to 1 and 0) and $\langle 000000100000010\rangle$ may be a transaction that includes a safety-critical instruction that sends Ethers (the 14th element is set to 1), which makes contracts leak Ethers to anyone.

5.4 Finding Zero-day Bugs in the Wild

We conducted an experiment to evaluate SMARTEST for finding unknown bugs from smart contracts in the wild. In November 2019, we collected 2,743 smart contracts with an open-source license from Etherscan[6] and ran SMARTEST (trained on CVE dataset) on the contracts with timeout 10 minutes for each contract. To ease our manual inspection on found bugs, we ran SMARTEST with an option to detect ERC20 standard violations only. We then manually inspected 142 automatically validated vulnerable transaction sequences (from 89 contracts) to judge the significance of found bugs. Below, we report two most significant bug patterns that SMARTEST found from 7 contracts, excluding benign and uncertain cases. We do not provide concrete addresses of vulnerable smart contracts to prevent abuse.

Pattern 1 (Mistakenly Named Constructor). Consider the code snippet below:

```
contract AToken {
  /* Constructor function */
  function BToken ( ) public {
    balance[msg.sender] = 10000000000;
    totalSupply = 10000000000; } ... }
```

where we deliberately modified the names of the contract and the function but included a part of the original comment ("Constructor function"). In old versions of Solidity ($\leq$ v.0.4.26), a function whose name is equal to the name of the contract was considered a constructor. Based on the comment in the code, we conjecture that the developer mistakenly named the constructor function. Due to this flaw, anyone can have `10000000000` tokens for free by invoking the `BToken` function. We found this type of vulnerabilities in 4 contracts with vulnerable transaction sequences of depths 3–4 generated by SMARTEST, by detecting violations of ERC20 standard invariants (Appendix C).

Pattern 2 (Unrestricted Token Transfer). Consider the `transferFrom` implementation below:

```
function transferFrom (address from, address to,
uint value) public returns (bool) {
  require (balance[from] >= value);
  require(balance[to] + value >= balance[to]);
  balance[from] -= value;
  balance[to] += value;
  return true; }
```

According to the description of the ERC20 standard interface [2], the `transferFrom` function should raise an exception if the original token holder (`from`) did not authorize a transaction sender (`msg.sender`). However, the above implementation does not impose any restrictions on transaction senders, i.e., there are no guard statements such as the one at line 29 of Figure 2. As a result, anyone can send money from one's account (`balance[from]`) to another's account (`balance[to]`) without any restrictions, if there are some balances in the `from`'s account. SMARTEST found these vulnerabilities with this pattern in 3 contracts by generating transactions of depth 1, each of which violates the specification of standrad `transferFrom` (Appendix C).

Note that, once the vulnerabilities described above are exploited in smart contracts that high market values, it can lead to considerable economic loss to existing token holders. For example, if the vulnerabilities in Pattern 1 are exploited, hackers can have large amounts for nothing. Moreover, due to these unrestricted token supplies, the market prices of the tokens may get lower, resulting in considerable economic loss to existing token holders.

5.5 Discussion

Limitations and Scope. As discussed in Section 5.2, our technique may not be effective when the training and test datasets contain different types of vulnerabilities. Another limitation is that our technique assumes a sufficient amount of vulnerable contracts for learning but such data may not be always readily available for some types of vulnerabilities.

Below we describe limitations and scope of our experiments in terms of covered vulnerabilities, and discuss potential extensions related to them. While we showed the effectiveness of our technique on six types of vulnerabilities, its effectiveness on vulnerabilities not covered in our experiments remains to be seen. In particular, in our evaluation, we did not consider vulnerabilities that require analysis of the interaction of multiple contracts to demonstrate the flaws (e.g., reentrancy). To support those types of vulnerabilities, we should be able to precisely handle external function calls

[6]https://etherscan.io/

(Section 4), possibly involving synthesis of unknown, interacting contracts. Moreover, to apply our technique to those types of vulnerabilities, we may need to extend our transaction representation method for identifying and prioritizing certain transactions that involve external function calls and are likely to reveal those types of vulnerabilities.

Exploitability of Vulnerabilities. While vulnerabilities found by SMARTEST include exploitable ones (e.g., batchOverflow vulnerability in CVE-2018-10299) but they would not be always immediately exploitable (e.g., CVE-2018-13085 where overflows can happen by a misuse of a contract owner rather than an arbitrary user). Nevertheless, we believe that our technique for effectively finding vulnerabilities (i.e., transaction sequences that violate safety conditions) is useful, because violations of safety conditions would be undesirable for safety-critical smart contracts. To precisely find immediately exploitable vulnerabilities, we need to formally specify the notion of exploitability in terms of logical formulas.

Threats to Validity. We describe potential sources of threats to validity that may be introduced in our experiments. Firstly, the benchmarks used in our experiments (443 contracts from CVE reports and 104 leaking and suicidal contracts from [14] and us) may not be representative and may be biased, although we tried hard for objective evaluation (e.g., collecting benchmarks from existing vulnerability databases, evaluating on trustful ground truths for vulnerabilities). Thus, when evaluated with other dataset whose regularities for vulnerable sequence patterns are rather different, results may be different. Secondly, comparing the vulnerability-finding capabilities among tools may be unfair in several aspects, despite our significant effort for a fair comparison (e.g., providing tool-specific constraints, giving more time budgets to other tools).

We describe concrete examples for the second point. As one example, we empirically found that modeling of leaking vulnerabilities differs in each tool. Specifically, we observed that MAIAN, TEETHER, MANTICORE, and ILF aim to find transaction sequences that leak Ethers to arbitrary addresses, assuming a test contract can have positive amounts of Ethers somehow (e.g., receiving Ethers from other killed contracts). Note that adopting this assumption may affect the effectiveness and the ground truth for vulnerabilities. As for the effectiveness aspect, following the assumption, a tool may be able to detect leaking vulnerabilities more quickly, since the vulnerabilities may be detected with shorter transactions without explicitly invoking `payable` functions. As for ground truth aspect, consider a simple contract without `payable` functions:

```
contract NoPayable {
  function sendEther () public {
    msg.sender.transfer(address(this).balance);}}
```

Observe that this contract has a leaking vulnerability with the assumption but does not have the vulnerability without the assumption (since the invariant `address(this).balance == 0` holds). MAIAN, TEETHER, MANTICORE, and ILF report the vulnerability for the above contract which does not have `payable` functions, while MYTHRIL does not. Regarding ground truths and SMARTEST's detection for leaking vulnerabilities, we followed the four tools' assumption, because we believe reporting issues related to improper access-controls would be beneficial rather than not reporting them. As another example for the second point, tools (TEETHER, ILF) that require additional inputs other than source code may yield better results if more sophisticated inputs are provided from users.

6 Related Work

Analysis of Smart Contracts. There is a large body of works on analysis of smart contracts, which we classify into four groups: symbolic execution [3,17,25,28,30,31,37,38], static analysis [9,10,16,39], formal verification [6,18,24,32,34,36], and fuzzing [19,22,27].

Symbolic execution, which SMARTEST builds upon, is perhaps the most popular approach for finding bugs in smart contracts. In particular, MYTHRIL [3], MANTICORE [30], MAIAN [31], TEETHER [25], and ETHBMC [17] are closely related to SMARTEST in that they also use symbolic execution and are able to generate vulnerable transaction sequences. MYTHRIL and MANTICORE are well-known and actively-maintained tools for finding a range of security vulnerabilities. MAIAN and TEETHER are tools for finding relatively high-level safety violations such as Ether-leaking and suicidal vulnerabilities. ETHBMC [17] focuses on precise modeling of EVM internals (e.g., cryptographic hash functions) for accurate analysis. Our focus is on improving the speed of symbolic execution with language models, where we believe our core idea is applicable to existing tools as well (possibly with some adjustments on transaction representation method for EVM bytecode). Other symbolic execution tools such as OYENTE [28], OSIRIS [37], and HONEYBADGER [38] do not automatically provide trace information of found bugs.

Static analysis and program verification have been also popular for smart contract security. Vandal [10], SECURIFY [39], and Ethainter [9] use Datalog-based static analysis for finding security vulnerabilities such as reentrancy. Slither [16] is a security checker that performs static analyses including data dependency analysis. ZEUS [24] is a verifier based on abstract interpretation. SMTCHECKER [6] and SOLC-VERIFY [18] are modular verification tools where each function is analyzed in isolation. VERISMART [36] automatically infers transaction invariants and uses them for precise verification. VERX [32] supports verification of temporal properties. *eThor* [34] is a provably sound verifier for EVM bytecode. However, unlike SMARTEST, these tools are inappropriate for generating transaction sequences due to abstractions.

Fuzzing is a simple yet effective method for analyzing smart contracts. ContractFuzzer [22] and REGUARD [27] are randomized fuzz testing tools for finding common vulnerabilities such as reentrancy. ILF [19] is an imitation learning-based fuzzer that aims to learn fuzzing policies from training sequences generated from symbolic execution.

Machine Learning for Symbolic Execution. There exist a few prior works that use machine learning to improve symbolic execution [11, 12, 26, 35, 40]. For example, MLB [26] uses machine learning to accelerate constraint solving in symbolic execution. To our knowledge, SMARTEST is the first that combines symbolic execution and language models to more effectively find vulnerabilities, although language models have been used in other contexts (e.g., code completion [33]).

7 Conclusion

We presented a new technique for effectively finding vulnerable transaction sequences in smart contracts. The key idea is to learn a statistical model from known vulnerable transaction sequences and use it to steer symbolic execution towards finding unknown vulnerabilities more effectively. We implemented the technique as a tool, SMARTEST, and demonstrated that SMARTEST is significantly more effective than existing tools for finding vulnerable transaction sequences.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Byron Williams, for their constructive comments that helped to improve this paper. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-51. This work was also supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337, (SW STAR LAB) Research on Highly-Practical Automated Software Repair). This work was also supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2019-0-01697, Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security). This research was also supported by the MSIT(Ministry of Science and ICT), Korea, under the ICT Creative Consilience program(IITP-2021-0-01819) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation). So was supported by the Korea University Graduate School Junior Fellow Research Grant.

References

[1] A $50 Million Hack Just Showed That the DAO Was All Too Human. https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/. Accessed: January 2021.

[2] ERC20 Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md. Accessed: January 2021.

[3] Mythril: a security analysis tool for EVM bytecode. https://github.com/ConsenSys/mythril. Accessed: January 2021.

[4] Solidity. https://docs.soliditylang.org/en/v0.8.0/. Accessed: January 2021.

[5] The Parity Wallet Hack Explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/. Accessed: January 2021.

[6] Leonardo Alt and Christian Reitwiessner. Smt-based verification of solidity smart contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 376–388, 2018.

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[8] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, 2007.

[9] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 454–469, 2020.

[10] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *CoRR*, abs/1809.03981, 2018.

[11] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. Automatically generating search heuristics for concolic testing. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1244–1254, 2018.

[12] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. Learning to accelerate symbolic execution via code transformation. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, 2018.

[13] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[14] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 530–541, 2020.

[15] Oscar Soria Dustmann, Klaus Wehrle, and Cristian Cadar. Parti: A multi-interval theory solver for symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 430–440, 2018.

[16] Josselin Feist, Gustavo Greico, and Alex Groce. Slither: A static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '19, page 8–15, 2019.

[17] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2757–2774, 2020.

[18] Ákos Hajdu and Dejan Jovanovic. solc-verify: A modular verifier for solidity smart contracts. *CoRR*, abs/1907.04262, 2019.

[19] Jingxuan He, Mislav Balunoviundefined, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 531–548, 2019.

[20] Frederick Jelinek. Interpolated estimation of markov source parameters from sparse data. In *Proc. Workshop on Pattern Recognition in Practice, 1980*, 1980.

[21] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA '15, page 177–187, 2015.

[22] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 259–269, 2018.

[23] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, 1st edition, 2000.

[24] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS*, 2018.

[25] Johannes Krupp and Christian Rossow. Teether: Gnawing at ethereum to automatically exploit smart contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 1317–1333, 2018.

[26] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. Symbolic execution of complex program driven by machine learning based constraint solving. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 554–559. IEEE, 2016.

[27] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: Finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68, 2018.

[28] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, 2016.

[29] B. K. Mohanta, S. S. Panda, and D. Jena. An overview of smart contract and use cases in blockchain technology. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–4, 2018.

[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 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189, 2019.

[31] Ivica Nikoliundefined, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 653–663, 2018.

[32] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 414–430, 2020.

[33] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 419–428, 2014.

[34] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. Ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 621–640, 2020.

[35] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. Neuro-symbolic execution: Augmenting symbolic execution with neural constraints. In *26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[36] S. So, M. Lee, J. Park, H. Lee, and H. Oh. Verismart: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 718–734, 2020.

[37] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 664–676, 2018.

[38] Christof Ferreira Torres, Mathis Steichen, and Radu State. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1591–1607, 2019.

[39] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 67–82, 2018.

[40] Sheng-Han Wen, Wei-Loon Mow, Wei-Ning Chen, Chien-Yuan Wang, and Hsu-Chun Hsiao. Enhancing symbolic execution by machine learning based solver selection. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2019.

Appendix

A Simplification Procedure in Section 3.1.2

For simplicity, assume FOL is defined by the grammar below:

$$F ::= \mathit{true} \mid \mathit{false} \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \forall y.x[y] = e \mid A \mid A^\circ \mid A^\bullet$$

F is a boolean constant ($\mathit{true}, \mathit{false}$), the application of a conjunction ($\wedge$), a disjunction ($\vee$), a negation ($\neg$) or a universal quantifier ($\forall$), or an atomic formula A possibly annotated with symbols ($\circ$ or $\bullet$), which provide hints for simplifying given constraints (Section 3.1, 3.1.2). An atomic formula A is a binary predicate applied to two terms (i.e., $A ::= e_1 \succ e_2$).

The simplification procedure is the following. Suppose a VC $F = F_1 \wedge \neg F_2$ is given where F is a conjunctive formula, F_1 is a state condition, and F_2 is a safety condition (Section 3.1.1); for State at line 13 of Algorithm 1, we consider $\mathit{State} \wedge \neg \mathit{false}$ (i.e., $F_1 = \mathit{State}, F_2 = \mathit{false}$). Let F_{ps} be the path conditions and Solidity-specific constraints in F_1, i.e., conjunctions of atomic formulas annotated with $\bullet$ symbols. We first collect a set of *necessary* variables X, which are needed to generate vulnerable transaction sequences. Concretely, we collect the initial set of variables $I = \mathsf{FV}(F_{ps}) \cup \mathsf{FV}(F_2)$, where $\mathsf{FV}(F')$ denotes the set of free variables in F'. Then, we iteratively collect all variables that may affect variables in I, until we reach a fixed point, i.e., $X = \mathsf{fix}(\lambda x.I \cup \mathsf{C}(F,x))$ where the "transfer function" $\mathsf{C} : \mathsf{FOL} \times \wp(\mathit{Var}) \to \wp(\mathit{Var})$ is defined below. Next, by iterating each conjunct F'' in F, we replace F'' by *true* if $\mathsf{FV}(F'') \not\subseteq X$ (F'' includes unnecessary variables). Finally, we remove symbols ($\circ$ or $\bullet$) in each atomic formula.

The function C is defined as follows:

$$\mathsf{C}(\mathit{true}, X) = X, \quad \mathsf{C}(\mathit{false}, X) = X$$

$$\mathsf{C}(F_1 \wedge F_2, X) = \mathsf{C}(F_1, X) \cup \mathsf{C}(F_2, X), \qquad \mathsf{C}(F_1 \vee F_2, X) = \mathsf{C}(F_1, X) \cup \mathsf{C}(F_2, X)$$

$$\mathsf{C}(\neg F, X) = \mathsf{C}(F, X), \quad \mathsf{C}(\forall y.x[y] = e, X) = \mathsf{C}(x[y] = e, X \setminus \{y\})$$

$$\mathsf{C}(e_1 \succ e_2, X) = \begin{cases} X \cup X' & \text{if } X \cap X' \neq \emptyset \text{ (where } X' = \mathsf{V}(e_1) \cup \mathsf{V}(e_2)) \\ X & \text{otherwise} \end{cases}$$

$$\mathsf{C}((x = e)^\circ, X) = \begin{cases} \mathsf{V}(e) \cup X & \text{if } x \in X \\ X & \text{otherwise} \end{cases}, \quad \mathsf{C}((e_1 \succ e_2)^\bullet, X) = \mathsf{C}(e_1 \succ e_2, X)$$

where $\mathsf{V}(e)$ means the set of variables in e. Given a formula F and a set of variables X, $\mathsf{C}(F,X)$ outputs a new set of variables $X'(\supseteq X)$ by adding variables in F into X, where the variables in F may affect some variables in X. The core part is $\mathsf{C}((x=e)^{\circ},X)$, where we collect $\mathsf{V}(e)$ only when x is identified as a necessary variable to be tracked. By constrast, in $\mathsf{C}(e_1 \succ e_2,X)$, we collect variables by considering information propagation at both sides (i.e., e_1 and e_2).

B Quantifier Elimination in Section 3.1.2

Given a verification condition F that may include universally quantified constraints, we obtain its quantified-free version F' using $\mathsf{QE} : \mathsf{FOL} \times \mathsf{FOL} \to \mathsf{FOL}$ (i.e., $F' = \mathsf{QE}(F,F)$):

$$\begin{gathered}
\mathsf{QE}(\mathit{true},F) = \mathit{true}, \qquad \mathsf{QE}(\mathit{false},F) = \mathit{false} \\
\mathsf{QE}(F_1 \wedge F_2,F) = \mathsf{QE}(F_1,F) \wedge \mathsf{QE}(F_2,F), \;\; \mathsf{QE}(F_1 \vee F_2,F) = \mathsf{QE}(F_1,F) \vee \mathsf{QE}(F_2,F) \\
\mathsf{QE}(\neg F',F) = \neg(\mathsf{QE}(F',F)) \\
\mathsf{QE}(\forall y.x[y] = e,F) = (x[y_1] = e) \wedge \cdots \wedge (x[y_n] = e) \text{ where } I_{F,x} = \{y_1,\cdots,y_n\} \\
\mathsf{QE}(e_1 \succ e_2,F) = e_1 \succ e_2
\end{gathered}$$

where $I_{F,x}$ denotes a set of index variables that are used as indices of x (or variables whose unprimed name is x, e.g., x', x'') in F. For example, when $F = x[p] = 3 \wedge x'[q] = 4$, $I_{F,x} = \{p,q\}$. Note that we do not define rules for the $(e_1 \succ e_2)^{\circ}$ and $(e_1 \succ e_2)^{\bullet}$ cases, because the symbols ($\circ$, $\bullet$) are removed after performing the property-focused simplification.

C Vulnerability Detection Rules (ERC20 Violation, Leaking, Suicidal)

ERC20 Violation. We implemented four harness functions equipped with rules for detecting ERC20 standard violations. We check these rules, by automatically inserting the test harness functions when predefined conditions are met and analyzing the augmented contracts.

A test harness for `transfer` functions checks: 1) whether the token sender's balance (e.g., `balance[msg.sender]`) is greater than or equal to `value` (an input parameter indicating the money to be sent) in case of successful transactions (i.e., returning *true*), 2) the token sender's balance is decreased by `value` (resp., not changed) in case of successful (resp., failing) transactions, and 3) the token receiver's balance (e.g., `balance[to]`) is increased by `value` (resp., not changed) in case of successful (resp., failing) transactions.

A test harness for `transferFrom` functions checks: 1) the token sender's balance (e.g., `balance[from]`) is greater than or equal to `value` in case of successful transactions, 2) the agent's allowance (e.g., `allowance[from][msg.sender]`) is greater than or equal to `value` in case of successful transactions, 3) the token sender's balance is decreased by `value` (resp., not changed) in case of successful (resp., failing) transactions, 4) the token receiver's balance (e.g., `balance[to]`) is increased by `value` (resp., not changed) in case of successful (resp., failing) transactions, and 5) the allowance is decreased by `value` (resp., not changed) in case of successful (resp., failing) transactions.

We also have two test harnesses for detecting violations of ERC20 invariants. The first harness checks whether the sum of balances between the two different accounts does not overflow, where the specification is an under-approximated one (e.g., it does not consider relationships among three account addresses). The second harness checks whether the balance of each account address is less than or equal to the total amount of supplied tokens (e.g., `totalSupply`).

Ether-leaking Vulnerability. Given a statement that sends Ethers to accounts, we report a leaking vulnerability if the contract leaks Ethers to an *untrusted* user and the amount of the leaked Ethers is greater than the amount of Ethers sent from the untrusted user. For example, given a statement `address(rcv).transfer(amount)` that sends `amount` Weis to `rcv`, we report a leaking vulnerability if the following safety condition can be violated: $\mathit{Trusted}[\mathit{rcv}] \vee \mathit{Invest}[\mathit{rcv}] \geq \mathit{money} \vee \mathit{money} = 0$. *Trusted* is an array that maps accounts to boolean values. We say an account `X` is *trusted* (resp., *untrusted*), if $\mathit{Trusted}[X]$ evaluates to *true* (resp., *false*) under a given satisfying assignment. A set of trusted addresses is defined as: hard-coded addresses in a source code, a message sender of initial transactions (i.e., a user that invokes constructors), `this` address, zero address (a constraint for ensuring an untrusted *rcv* is not a zero address, considering realistic scenarios), and address-typed parameters in transactions sent from trusted accounts [19]. *Invest* is an array that tracks the amount of Ethers invested from each user.

Suicidal Vulnerability. Given a statement that deactivates contracts, we report a suicidal vulnerability if the statement can be executed by untrusted users. For example, given a `selfdestruct(...)` statement, we report a vulnerability if the safety condition $\mathit{Trusted}[\mathit{msg.sender}]$ can be violated.

D Mutation Patterns (Leaking, Suicidal)

We describe mutation patterns for injecting likely leaking and suicidal vulnerabilities into seed contracts. These patterns aim to cause improper access controls (e.g., allowing anyone to access safety-critical statements), since one typical root reason for those vulnerabilities is based on them.

Pattern 1 is to negate conditions in modifiers (e.g., `onlyOwner`) that check ownership of contracts (e.g., changing `require(msg.sender==owner)` to `require(msg.sender != owner)`). Pattern 2 is to remove modifiers for checking ownership in functions, which include statements that send Ethers (e.g., `transfer`) or kills contracts (e.g., `selfdestruct`). Pattern 3 is to remove modifiers for checking ownership in functions (e.g., `transferOwnership`) being in charge of transferring ownership.

The statistics for 54 constructed benchmarks are as follows: Pattern 1 (20), Pattern 2 (17), and Pattern 3(17).