SmarTrim: Symbolic Execution for Smart Contracts Powered by Redundant Transaction-Sequence Pruning

HYEGEUN SONG, Korea University, Republic of Korea
JISEONG HAN, Korea University, Republic of Korea
SUNBEOM SO*, Korea University, Republic of Korea

We present SmarTrim, a new symbolic execution technique for detecting vulnerabilities in smart contracts. Smart contracts require rigorous safety validation since flaws in them can cause significant financial loss. Numerous symbolic execution techniques, which generate vulnerable transaction sequences to trigger and help understand vulnerabilities, have been extensively studied to enhance the security and safety of smart contracts. However, their performance remains unsatisfactory due to the extremely large search space for transaction sequences. To mitigate this issue, SmarTrim introduces a novel technique that safely reduces the search space by detecting and pruning redundant transaction sequences. Experimental results show that SmarTrim greatly outperforms eleven state-of-the-art analyzers in detecting critical vulnerabilities in real-world smart contracts.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: smart contract, symbolic execution, safe pruning

ACM Reference Format:
Hyegeun Song, Jiseong Han, and Sunbeom So. 2026. SmarTrim: Symbolic Execution for Smart Contracts Powered by Redundant Transaction-Sequence Pruning. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE046 (July 2026), 24 pages. https://doi.org/10.1145/3797074

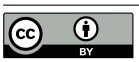
1 Introduction

Secure smart contracts are essential for building reliable blockchain ecosystems. Smart contracts–programs running on blockchains–are gaining increasing popularity, as they can reliably automate agreements between untrusted parties, eliminating the need for intermediaries. Unfortunately, however, smart contracts are frequently targeted by attackers as they manipulate valuable assets, leading to tremendous financial losses (e.g., [1, 35, 58, 60]) once exploited. As a result, the demand for techniques to ensure the safety of smart contracts is steadily increasing.

In this paper, we present SmarTrim, a new symbolic execution technique for Ethereum smart contracts. Symbolic execution has been actively used to find critical flaws in smart contracts as it can complement shortcomings of other program analysis approaches. For example, unlike verification (e.g., [46, 57, 63, 66, 70, 75]) or static bug-finding (e.g., [30, 36, 40, 49, 51, 52, 72, 77, 86]), symbolic execution can generate transaction sequences (i.e., function invocation sequences with concrete arguments), helping understand under what conditions vulnerabilities can occur. Also, unlike

*Corresponding author

Authors' Contact Information: Hyegeun Song, Korea University, Seoul, Republic of Korea, hyegeun_song@korea.ac.kr; Jiseong Han, Korea University, Seoul, Republic of Korea, jiseong104@korea.ac.kr; Sunbeom So, Korea University, Seoul, Republic of Korea, sunbeom_so@korea.ac.kr.



ACM 2994-970X/2026/7-ARTFSE046
https://doi.org/10.1145/3797074

fuzzing (e.g., [31, 41, 59, 64, 71, 78]), it is more likely to produce bug-triggering inputs that explore complicated paths, aided by modern advanced SMT solvers.

Prior Work. Numerous symbolic execution approaches for smart contracts have been proposed recently (e.g., [28, 34, 47, 54, 55, 65, 76, 81–85]), but they continue to suffer from performance issues. In particular, most existing techniques are not able to effectively handle transaction sequences that are *redundant* with previously analyzed sequences. For example, path prioritization techniques [54, 65], which look for promising sequences first, do not always ensure that such redundant sequences are deprioritized. Approaches for pruning unnecessary sequences have also been proposed; they discard sequences without read-after-write (RAW) relationships between transactions [81, 82]. However, RAW information alone is insufficient to eliminate diverse redundant sequences (e.g., Examples 1 and 2 in Section 2.1). More detailed discussions are in Section 2.2.

Our Work. We propose SmarTrim to address the limitations of existing approaches. The key differentiator is a domain-specific technique for identifying and pruning a large number of redundant transaction sequences during symbolic execution of smart contracts. In Solidity smart contracts, only global variables persist across transactions (i.e., function invocations from external users). Based on this property, we introduce *subsumption condition*, a novel criterion to determine whether a candidate sequence is redundant with previously explored sequences. Specifically, the subsumption condition, denoted $\mathsf{subsumed}(s, v)$, holds iff every global state (i.e., mapping from global variables to values) reachable by a candidate sequence s is also reachable by some previously validated sequence v. SmarTrim prunes s whenever $\mathsf{subsumed}(s, v)$ holds. Our technique guarantees to safely reduce the search space: any vulnerability discoverable by extending a pruned sequence can also be detected by extending some remaining sequence.

Results. Experimental results show that SmarTrim is highly effective at finding vulnerabilities in smart contracts. We implemented SmarTrim for Solidity [16], the de facto standard language for developing Ethereum smart contracts. We compared SmarTrim with eleven analyzers on three datasets containing four types of widely studied security-critical vulnerabilities: ether-leak, suicidal, integer over/underflow, and reentrancy. The selected competing tools employ different analysis methods: six symbolic executors (SmarTest [65], Mythril [11], LENT-SSE [84], AChecker [37], Sailfish [28], SliSE [76]), four fuzzers (Smartian [31], ConFuzzius [73], RLF [71], EF/CF [62]), and one static analyzer (Slither [33]). The three benchmarks consist of 814 smart contracts that are mostly real-world and collected from multiple prior studies [2, 28, 31, 53, 65, 67, 86]. The results show that SmarTrim far surpasses eleven analyzers in vulnerability detection.

Contributions. We summarize our contributions.

- We present a new technique for finding vulnerabilities in smart contracts. The key idea is to detect and prune redundant transaction sequences during symbolic execution, based on reachable global states.
- We demonstrate the effectiveness of SmarTrim through extensive comparative experiments with 814 smart contracts and 11 analyzers [11, 28, 31, 33, 37, 62, 65, 71, 73, 76, 84].
- To support open science, we provide the source code of SmarTrim and datasets on both Zenodo [68] and GitHub [69].

2 Overview

2.1 Motivating Examples

Figure 1 shows the `Trabet_Coin` contract [21, 22] written in Solidity [16]. The contract has five global variables (lines 2–4). `owner` stores the account address of the contract's owner. `crowdsaleAgent` holds the address of the contract's admin. `totalSupply` refers to the total amount of

```
1  contract Trabet_Coin {
2    address owner; address crowdsaleAgent; uint totalSupply;
3    mapping(address=>uint) balance;
4    mapping(address=>mapping(address=>uint)) allowance;
5
6    constructor() { owner = msg.sender; totalSupply = 10000; balance[owner] = totalSupply; }
7
8    function mint(address t, uint v) public {
9      require(msg.sender == crowdsaleAgent);
10     balance[t] += v;
11     totalSupply += v;
12   }
13
14   function burn(uint v) public {
15     require(balance[msg.sender] >= v);
16     balance[msg.sender] -= v;
17     totalSupply -= v;
18   }
19
20   function burnFrom(address from, uint v) public {
21     require(balance[from] >= v);
22     require(v <= allowance[from][msg.sender]);
23     balance[from] -= v;
24     allowance[from][msg.sender] -= v;
25     totalSupply -= v;
26   }
27
28   function approve(address spender, uint v) public {
29     allowance[msg.sender][spender] = v;
30   }
31
32   function setOwner(address o) public {
33     require(msg.sender == owner);
34     owner = o;
35   }
36
37   function setCsAgent(address c) public {
38     require(msg.sender == owner);
39     crowdsaleAgent = c;
40   }
41 }
```

Fig. 1. `Trabet_Coin` contract (simplified for presentation).

circulating tokens as an unsigned 256-bit integer (`uint`). `balance` is a mapping that stores the number of tokens per account. `allowance` is a two-dimensional mapping; `allowance[A][B]` denotes the number of tokens allowed to be used by a spender (`B`) on behalf of the token holder (`A`).

A transaction, which can change the blockchain's state, is triggered by invoking a function, where the caller is referred to as the transaction message sender (`msg.sender`). For example, calling the constructor (line 6) triggers an initial transaction, which deploys the contract on the blockchain, assigns the contract's ownership to the caller, and sets both `totalSupply` and `balance[owner]` (the owner's token). The function `mint` can be executed by `crowdsaleAgent` only (line 9) and issue additional tokens (lines 10, 11). `burn` destroys the sender's balance and reduces `totalSupply` accordingly (lines 16, 17). `burnFrom` (lines 20–26) allows an agent (`msg.sender`) to burn `v` tokens, on behalf of the original token holder (`from`). With `approve` (lines 28–30), token holders (`msg.sender`) can authorize `spender` to use up to `v` tokens on their behalf. The owner can transfer ownership via `setOwner` (lines 32–35) and delegate admin rights via `setCsAgent` (lines 37–40).

The contract has integer under/overflow bugs at lines 10, 11, 17, and 25, but generating transaction sequences to detect them is not trivial. In particular, the underflow at line 25 can be exposed by a

sequence of at least four transactions (excluding the initial transaction t_0), such as $t_0 \cdot t_1 \cdot t_2 \cdot t_3 \cdot t_4$ where t_0 : `constructor( )` with `msg.sender = A`,

t_1 : `setCsAgent(B)` with `msg.sender = A`, t_2 : `mint(B, MAX)` with `msg.sender = B`
t_3 : `approve(A, MAX)` with `msg.sender = B`, t_4 : `burnFrom(B, MAX)` with `msg.sender = A`

Here, `A` and `B` are the addresses of the contract's owner and admin, and `MAX` $= 2^{256} - 1$ (the maximum value of the `uint` type). Observe that `totalSupply` overflows to `9999` in t_2 and it wraps around to `10000` due to the underflow at line 25 in t_4.

Unfortunately, most smart contract analyzers are not effective here. For example, given the 210-line original contract [22] for Figure 1, ConFuzzius [73], LENT-SSE [84], Mythril [11] and Smartian [31] failed to find the bug at line 25 in our experiments (Section 6).

In contrast, SmarTrim discovers the bug at line 25 quickly, within 2 minutes. It does so by detecting and pruning a number of transaction sequences that are semantically redundant with previously analyzed sequences, in terms of reachable global states. We provide examples of such redundant transaction sequences.

Example 1. Non-state-changing Transactions. Suppose we analyze $p : t_0 \cdot t_1$ after symbolic execution on $q : t_0$, where

t_0 : `constructor( )` with `msg.sender = A`, t_1 : `burnFrom(C, n)` with `msg.sender = B`,

and the symbolic arguments (`A`, `B`, `C`, and `n`) represent all possible concrete values that match the types of `msg.sender` or each function parameter. Note that p is redundant with q with respect to reachable global states (i.e., mappings from global variables to their values). To explain why, since all mapping elements of type `uint` are initially zeros (e.g., $\forall$`x, y.allowance[x][y] = 0`) in Solidity [16], line 22 of t_1 in p is passed through only when `n = 0`, resulting in no changes to any global variables, including `allowance` and `balance`. Consequently, we can safely remove p's subsequent sequences from the search space, without compromising bug detection – that is, any vulnerability that can be detected by extending $t_0 \cdot t_1$ can also be discovered by extending t_0.

Example 2. Subsumed Transactions. Suppose we perform symbolic execution on $p : t_0 \cdot t_1 \cdot t_2$ after verifying $q : t_0 \cdot t_1$, where t_0 : `constructor( )` with `msg.sender = A`,

t_1 : `setOwner(o1)` with `msg.sender = B`, t_2 : `setOwner(o2)` with `msg.sender = C`,

and `A, B, C, o1,` and `o2` are symbolic arguments. Observe that p is redundant with q, in that every global state inducible by p is also inducible by q. This holds because: (1) in both t_1 and t_2, the only change to the global state occurs in `owner`, and (2) `o1` and `o2`, which are assigned to `owner`, represent the same set of concrete addresses. That is, given p, the behavior of t_2 is subsumed by that of the preceding transaction t_1.

Example 3. Interchangeable Transactions. Suppose we analyzed $q : t_0 \cdot t_1 \cdot t_2$ and encountered $p : t_0 \cdot t_2 \cdot t_1$, where t_0 : `constructor( )` with `msg.sender = A`,

t_1 : `burn(v1)` with `msg.sender = B`, t_2 : `setOwner(o2)` with `msg.sender = C`,

and `A, B, C, v1,` and `o2` are symbolic arguments. Despite the difference in the order between t_1 and t_2, the symbolic execution on p and q will produce the same set of global states. To see why, the variables (`balance`, `totalSupply`) defined in t_1 are neither redefined nor constrained in t_2; and similarly, the variable (`owner`) defined in t_2 is neither redefined nor constrained in t_1. That is, the global state changes caused by t_1 and t_2 are disjoint. As a result, p is redundant with q.

```
1  contract Example {
2    bool flag; uint x;
3
4    constructor ( ) { flag = false; x = 0; }
5
6    function setFlag (bool b) { flag = b; }
7    function setX (uint y) { x = y; }
8    function setX10 ( ) { x = 10; }
9    function f ( ) { require(flag); assert(x != 10); }
10 }
```

Fig. 2. A contract for illustrating SmarTrim approach.

2.2 How to Prune Redundant Sequences

We illustrate how SmarTrim works on the contract in Figure 2. Consider symbolic transactions:

$$t_0\colon \texttt{constructor( )}, \quad t_1\colon \texttt{setFlag(b)}, \quad t_2\colon \texttt{setX(n)}, \quad t_3\colon \texttt{setX10( )}, \quad t_4\colon \texttt{f( )}$$

where we omitted transaction senders (`msg.sender`) for simplicity. Suppose our goal is to find a vulnerable sequence to trigger the assertion violation at line 9 in Figure 2, such as $t_0 \cdot t_1 \cdot t_2 \cdot t_4$.

To quickly find vulnerable sequences, during symbolic execution, SmarTrim detects and prunes transaction sequences that are redundant with previously analyzed sequences. For example, if a sequence $q : t_0 \cdot t_2$ has been validated, SmarTrim determines that a new sequence $p : t_0 \cdot t_3$ is redundant with q, because the following *subsumption condition* holds:

$$A_p \subseteq A_q \tag{1}$$

where A_p and A_q refer to the sets of all global states that are reachable after the executions of p and q. That is, $A_p = \{[flag \mapsto false, x \mapsto 10]\}$, and $A_q = \{[flag \mapsto false, x \mapsto 0], \cdots, [flag \mapsto false, x \mapsto 2^{256} - 1]\}$. However, directly checking (1) is computationally expensive in general, as it requires enumerating all global states [24] reachable by each sequence; for example, computing A_q may need up to 2^{256} invocations of an SMT solver (e.g., [32]). Thus, we instead verify a condition (2) equivalent to (1), which can be checked with a single call to an SMT solver:

$$\psi : \exists x'.\ \underbrace{flag = false \wedge x' = 0 \wedge x = 10}_{\phi_p} \rightarrow \exists x', y.\ \underbrace{flag = false \wedge x' = 0 \wedge x = y}_{\phi_q} \text{ is valid} \tag{2}$$

Here, ϕ_p (resp., ϕ_q) represents the state condition whose satisfying models correspond to states inducible by p (resp., q). The primed variable x' in ϕ_p (resp., ϕ_q) denotes x redefined by t_3 (resp., t_2). The existentially quantified variables are responsible for projecting the states for ϕ_p and ϕ_q, only onto the global variables. Observe that, for any global state a, we have $a \in A_p \Leftrightarrow a \models \exists x'.\phi_p$ and $a \in A_q \Leftrightarrow a \models \exists x', y.\phi_q$. Thus, (1) holds iff (2) holds. We formalize this property in Proposition 3.1. Since (2) indeed holds (i.e., ψ is valid), SmarTrim concludes that p is redundant with q, thereby eliminating subsequent sequences of p from the search space. In the supplement A, we provide examples that illustrate how our approach can be applied to more complex contracts involving mappings, arrays, and structures, beyond those with simple scalar variables.

Comparison with Existing Approaches. SmarTrim offers a distinct contribution to symbolic execution of smart contracts, i.e., a new technique for pruning a range of redundant transaction sequences, based on reachable global states. Several prior studies have proposed pruning unnecessary sequences without read-after-write (RAW) relationships [81, 82], but not all redundant sequences can be detected by relying solely on RAW relationships (e.g., Examples 1 and 2 in Section 2.1). Also, unlike approaches that prune paths irrelevant to specific targeted vulnerabilities [55, 76, 83], our technique is generally applicable in a vulnerability-agnostic manner. EthRacer [47] prunes infeasible sequences, but unlike SmarTrim, it cannot eliminate feasible yet redundant sequences

(Section 8). Other symbolic execution methods for smart contracts aim to boost performance along orthogonal directions, focusing on areas unrelated to semantic redundancy between paths: prioritizing likely paths [54, 65], summary-based technique to avoid constraint recomputation [34], state merging [28], parallel symbolic execution [85], and skipping path feasibility checks to reduce SMT solving time [84].

Our work also makes a unique contribution compared to symbolic execution techniques from conventional program domains (e.g., C or Java). For example, similar to ours, several studies [56, 79, 80] aim to eliminate redundant program paths. However, they focus on eliminating paths sharing identical path suffixes, whereas we aim to detect redundancy even when transaction suffixes differ (e.g., Examples 1-3 in Section 2.1). Section 8 provides more detailed discussions.

3 SmarTrim Approach

This section formalizes: basic symbolic execution procedure (Section 3.1), and its enhancement with safe pruning (Section 3.2). We first define a smart contract and introduce terminology based on prior related work [65, 66].

Contract. We present our technique for a core subset [65, 66] of Solidity [16]. A contract $c = (G, F)$ consists of a set of global variables (G) and a set of functions (F). A function $f(x)\{s\} \in F$ is comprised of f (function name), x (input parameter), and s (function body). We write f_0 to denote the name of a constructor function. We assume a set of statements S is defined by a simple grammar:

$$s \rightarrow a \mid if\, e\, s_1\, s_2 \mid s_1; s_2, \qquad a \rightarrow x := e \mid x[y] := e \mid assume(e) \mid assert^l(e)$$

A statement s is an atomic statement a, an if-statement, or a sequence of statements. Function call statements are inlined and loops are unrolled during preprocessing (Section 5), so they do not appear in the grammar. An atomic statement is an assignment for scalar variables ($x := e$) or one-dimensional mapping elements ($x[y] := e$), an *assume* statement, or an *assert* statement. In the grammar, e represents conventional expressions, and we assume that the outermost expressions in *assume* and *assert* are bool-typed. *assume* is responsible for modeling branch conditions (e.g., true/-false branches of if-statements) when generating function paths (explained below). $assert^l(e)$ does not alter program semantics and merely specifies a safety property e to verify at program point l.

Terminology. A *function path*, denoted $p = (f, x, a)$, refers to a sequence of atomic statements (a), which models the execution path within a function $f(x)\{s\}$. A *transaction*, denoted $t = (id, p)$, is a function path p labeled with a unique transaction identifier id. A *transaction sequence*, denoted $s = t_0 \cdot t_1 \cdots t_n$, is a chain of transactions. If s triggers a vulnerability (i.e., violates a condition in *assert*) at the last transaction t_n, we say s is a vulnerable transaction sequence. t_0 refers to an initial transaction constructed from a constructor function path; that is, given $t_0 = (id, (f, x, a))$, $f = f_0$.

3.1 Basic Symbolic Execution

Algorithm 1 shows the architecture of SmarTrim. Its goal is to find as many vulnerable transaction sequences as possible. The input is a contract c written in Solidity. The output is a vulnerability report R that provides a vulnerable transaction sequence per assertion label. While running the algorithm, the workset W maintains a set of candidate transaction sequences to validate. We assume lines 13–14 are ignored in our basic approach.

At line 1, we generate a set of function paths P of c, and at line 2, construct a set of constructor paths P_0. At line 3, W is initialized with a set of initial transactions. At line 4, R is initialized with $\lambda l.\bot$, meaning that no vulnerable transaction sequences have been detected at this stage. At line 5, we set V, a set of validated transaction sequences, to the empty set.

Algorithm 1 SmarTrim algorithm

Input: A Solidity smart contract c
Output: A vulnerability report R

1: $P \leftarrow$ a set of function paths from c
2: $P_0 \leftarrow \{(f, x, a) \mid (f, x, a) \in P, f = f_0\}$
3: $W \leftarrow \{(id, p) \mid p \in P_0, \text{ new } id\}$
4: $R \leftarrow \lambda l.\bot$
5: $V \leftarrow \emptyset$
6: **repeat**
7: Pick a candidate sequence s from W
8: $W \leftarrow W \setminus \{s\}$
9: $s' \leftarrow$ **if** $s = t_0$ **then** t_0 **else** s'' where $s = s'' \cdot t_n$ ▷ s': the longest prefix of s
10: $\phi', \Pi' \leftarrow \mathsf{GenerateVC}(s')$ ▷ § 3.1
11: **if** $\neg\mathsf{SAT}(\phi')$ **then** ▷ s' is infeasible
12: $W \leftarrow W \setminus \{w \in W \mid w = s' \cdot t\}$
13: (+) **else if** $s' \notin V \wedge \exists v \in V.\text{subsumed}\,(s', v)$ **then** ▷ s' is redundant, § 3.2
14: (+) $W \leftarrow W \setminus \{w \in W \mid w = s' \cdot t\}$
15: **else**
16: $\phi, \Pi \leftarrow \mathsf{GenerateVC}(s)$
17: **for each** $(l, VC) \in \Pi$ **do**
18: **if** $R(l) = \bot \wedge \mathsf{SAT}(VC)$ **then**
19: $R \leftarrow R \cup \{l \mapsto (s, \mathsf{model}(VC))\}$
20: $V \leftarrow V \cup \{s'\}$
21: $W \leftarrow W \cup \{s \cdot (id, p) \mid p \in P \setminus P_0, \text{ new } id\}$
22: **until** $W = \emptyset$ or $\forall l.R(l) \neq \bot$ or timeout
23: **return** R

We enter the repeat-until loop (lines 6–22) that iteratively selects a candidate sequence and validates it using symbolic execution. We first choose a candidate sequence s (line 7) and delete it from W (line 8). At line 9, we compute s', the longest prefix of s: technically, $s' = t_0$ if $s = t_0$, and s' is the longest proper prefix of s otherwise. At line 10, we perform symbolic execution on s', generating two kinds of constraints: ϕ' and Π'. ϕ' stands for a state condition whose satisfying model represents a state reached at the end of s'. Π' is a set of verification conditions (VCs) per label l, which must be verified to see whether s' is a vulnerable transaction sequence that violates the safety condition at l. If s' is an infeasible transaction sequence (line 11), we remove candidate sequences beginning with s' from the search space (line 12). If s' is feasible (line 15), we produce VCs, denoted Π, for s (line 16). Each VC is examined through the for-loop at lines 17–19; if s is the first sequence that violates the safety condition at l (line 18), we update R accordingly (line 19). At line 20, we add s' to V (the set of verified sequences). At line 21, we generate new candidate transaction sequences by extending s with new transactions (other than initial transactions), and add them to W. The repeat-until loop iterates until W becomes empty, every safety condition is violated ($\forall l.R(l) \neq \bot$), or the preset time limit is reached (line 22). At line 23, we return R.

Note that our basic algorithm already adopts an optimization: rather than eagerly discarding infeasible sequences when generating new candidates (line 21) as in [65], we eliminate infeasible sequences lazily whenever they are chosen as candidate sequences (line 7). This way, it can reduce unnecessary calls to an SMT solver for sequences that would not be analyzed within a time budget.

Symbolic Execution of Transaction Sequences. We define GenerateVC (lines 10 and 16 in Algorithm 1), which generates a state condition and verification conditions by symbolically executing a sequence s. Let us first introduce sp and T, the symbolic executors for atomic statements and transactions, respectively.

Given a current state condition ϕ and (labeled) verification conditions Π, sp transforms them according to the execution semantics of each atomic statement. sp follows a standard definition of the strongest postcondition predicate transformer [29]:

$$\begin{array}{rcl}
\mathsf{sp}(x := e)(\phi, \Pi) & = & (\phi[x'/x] \wedge x = e[x'/x], \Pi) \\
\mathsf{sp}(x[y] := e)(\phi, \Pi) & = & (\phi[x'/x] \wedge x = x'\langle y \triangleleft e[x'/x]\rangle, \Pi) \\
\mathsf{sp}(\mathit{assume}(e))(\phi, \Pi) & = & (\phi \wedge e, \Pi) \\
\mathsf{sp}(\mathit{assert}^l(e))(\phi, \Pi) & = & (\phi, \{(l, \phi \wedge \neg e)\} \cup \Pi) \\
\mathsf{sp}(a_1; \cdots; a_n)(\phi, \Pi) & = & \mathsf{sp}(a_n) \cdots \mathsf{sp}(a_1)(\phi, \Pi)
\end{array}$$

$\phi[x'/x]$ denotes a formula produced by replacing the free variable x by a fresh primed variable x'. Here, x' denotes x before being updated by an assignment. The notation $x\langle y \triangleleft e\rangle$ represents a new array obtained by replacing the element at index y with e from the array x. Given an assertion, we generate a verification condition $\phi \wedge \neg e$ whose satisfiability indicates that the safety condition e can be violated under the state described by ϕ.

On top of sp, we define T that performs symbolic execution on a transaction $t = (id, (f, x, a))$:

$$\mathsf{T}(t)(\phi, \Pi) = (\phi'', \Pi'')$$

where $\phi'' = \textsc{Rename}_{id}(\phi')$, $\Pi'' = \{(l, \textsc{Rename}_{id}(\psi')) \mid (l, \psi') \in \Pi'\}$, and $(\phi', \Pi') = \mathsf{sp}(a)(\phi \wedge x^e = x, \Pi)$. Here, x^e is a variable that stores the x's value at the entry of the transaction. To distinguish local variables (including primed global variables) with identical names across different transactions, $\textsc{Rename}_{id}$ appends the current transaction's identifier (id) to local variable names. For example, given a set of global variables $G = \{a, b\}$ and $\phi : a = a' + x_2 \wedge b = a + b'$, $\textsc{Rename}_3(\phi)$ outputs $a = a'_3 + x_2 \wedge b = a + b'_3$.

Finally, we define GenerateVC (lines 10 and 16 in Algorithm 1):

$$\mathsf{GenerateVC}(t_0 \ \cdots \ t_n) = \mathsf{T}(t_n)(\phi', \emptyset)$$

ϕ' is the state condition for the sequence's prefix: $(\phi', \Pi') = \mathsf{T}(t_{n-1}) \circ \cdots \circ \mathsf{T}(t_0)(\bigwedge_{g \in G} \mathsf{init}(g), \emptyset)$. Observe that we discard Π', the verification conditions (VCs) generated from the prefix, in order to avoid duplicated VC checks. The role of init is to assign a default value to each variable based on its type. For example, $\mathsf{init}(g) = (g = \mathit{false})$ if g has type `bool`, $\mathsf{init}(g) = (g = 0)$ if g has type `uint256`, and $\mathsf{init}(g) = (g = \mathsf{K}_0)$ if g has type `mapping(address => uint256)` where K_0 denotes a constant array that returns 0 for any account.

3.2 Symbolic Execution with Pruning

We present the key technical contribution of this paper: effectively detecting and pruning redundant transaction sequences, leveraging reachable global states.

Subsumption Condition. In Solidity smart contracts, only global variables (i.e., variables stored in contract storage [16]) persist across transactions. By contrast, local variables exist only during function execution and thus are discarded at the end of each transaction. Based on this property, we define the predicate subsumed (line 13 in Algorithm 1).

Let G be a set of global variables. Let ST be a symbolic executor that generates the state condition for a sequence $s = t_0 \cdots t_n$:

$$\mathsf{ST}(s) = \mathsf{ST}'(t_n) \circ \cdots \circ \mathsf{ST}'(t_0)(\bigwedge_{g \in G} \mathsf{init}(g))$$

where ST' produces the state condition for a transaction t; that is, $\mathsf{ST}'(t)(\phi) = \phi'$ where $(\phi', -) = \mathsf{T}(t)(\phi, \emptyset)$. Let $A(s)$ be the set of global states reachable by s:

$$A(s) = \{\mathsf{Project}_G(m) \mid m \models \mathsf{ST}(s)\} \tag{3}$$

where $\mathsf{Project}_G(m) = \lambda x \in G.m(x)$. Let $v \in V$ be a validated sequence during symbolic execution (Algorithm 1). Then, we define the subsumption condition $\mathsf{subsumed}(s, v)$ as:

$$A(s) \subseteq A(v) \tag{4}$$

That is, we say s is subsumed by v iff all global states reachable via s are also reachable via v.

Checking Subsumption Condition. Directly deciding (4) is computationally expensive in general [24]. Specifically, computing every reachable global state for each sequence may require numerous calls to an off-the-shelf SMT solver (e.g., Section 2.2). To address this, we reformulate (4). Let $\mathsf{FV}(F)$ be the set of free variables of a first-order logic formula F. Let $\mathbf{X}$ and $\mathbf{Y}$ be the sets of local variables in $\mathsf{ST}(s)$ and $\mathsf{ST}(v)$, respectively: $\mathbf{X} = \mathsf{FV}(\mathsf{ST}(s)) \setminus G$ and $\mathbf{Y} = \mathsf{FV}(\mathsf{ST}(v)) \setminus G$. Given $\mathbf{X} = \{x_1, \cdots, x_n\}$ and $\mathbf{Y} = \{y_1, \cdots, y_m\}$, let us write $\exists \mathbf{X}.P$ and $\exists \mathbf{Y}.Q$ to represent $\exists x_1, \cdots, x_n.P$ and $\exists y_1, \cdots, y_m.Q$, respectively. We introduce a condition equivalent to (4), which can be checked with a single call to an SMT solver:

$$\exists \mathbf{X}.\mathsf{ST}(s) \rightarrow \exists \mathbf{Y}.\mathsf{ST}(v) \text{ is valid} \tag{5}$$

The following proposition formalizes the logical equivalence between (4) and (5).

Proposition 3.1 (Equivalence). $A(s) \subseteq A(v) \iff \models \exists \mathbf{X}.\mathsf{ST}(s) \rightarrow \exists \mathbf{Y}.\mathsf{ST}(v)$

Proof. Proof based on equational rewriting. $A(s) \subseteq A(v)$ holds iff

$$\forall m.\ m \in A(s) \implies m \in A(v) \tag{6}$$

By Equation (3), we have

$$m \in A(s) \Leftrightarrow m \models \exists \mathbf{X}.\mathsf{ST}(s), \quad m \in A(v) \Leftrightarrow m \models \exists \mathbf{Y}.\mathsf{ST}(v) \tag{7}$$

Based on (7), we rewrite (6):

$$\forall m.\ m \models \exists \mathbf{X}.\mathsf{ST}(s) \implies m \models \exists \mathbf{Y}.\mathsf{ST}(v) \tag{8}$$

By the semantics of implication [29], we rewrite (8):

$$\forall m.\ m \models (\exists \mathbf{X}.\mathsf{ST}(s) \rightarrow \exists \mathbf{Y}.\mathsf{ST}(v)) \tag{9}$$

Observe that (9) and (6) are equivalent, as (9) is derived from (6) via equational rewriting. □

Equation (4) and Proposition 3.1 together yield a derived definition of subsumed, used in the implementation of SmarTrim.

Definition 3.2 (Subsumption Condition). *s is subsumed by v in terms of reachable global states:* $\mathsf{subsumed}(s, v) \iff A(s) \subseteq A(v)$. *Equivalently,* $\mathsf{subsumed}(s, v) \iff \models \exists \mathbf{X}.\mathsf{ST}(s) \rightarrow \exists \mathbf{Y}.\mathsf{ST}(v)$.

Symbolic Execution with Pruning. SmarTrim performs pruning by adding lines 13 and 14 to Algorithm 1. If s' is redundant with some validated sequence other than itself (line 13), we remove every subsequent sequence of s' from W (line 14).

Pruning Safety. Our pruning at line 13 is safe in the following sense: if a sequence s is eliminated by our pruning, any vulnerability that can be discovered by extensions of s can still be discovered by extensions of some $v \in V$. The following theorem formalizes this property.

Theorem 3.3 (Pruning Safety). *Given two transaction sequences s and v, suppose* $\mathsf{subsumed}(s, v)$ *holds. Suppose also we have a transaction sequence $s' = t'_1 \cdots t'_n$ where t'_n contains an assertion statement $\mathsf{assert}^l(e)$. If e can be falsified by executing $s \cdot s'$, it can also be falsified by executing $v \cdot s'$.*

To prove the theorem, let $\sigma : \mathsf{Dom}(\sigma) \to \mathsf{Ran}(\sigma)$ be a variable substitution by the symbolic execution of assignments in a transaction t. $\mathsf{Dom}(\sigma)$ is a set of variables to rename and $\mathsf{Ran}(\sigma)$ is a set of corresponding primed variables, that is, $\mathsf{Dom}(\sigma) = \mathsf{Def}(t)$ and $\mathsf{Ran}(\sigma) = \{x' \mid x \in \mathsf{Def}(t)\}$, where $\mathsf{Def}(t)$ is the set of global variables that may be defined via assignments in t. Given a formula F, we write $F\sigma$ to represent the application of σ to F. For example, given $F : x + 1 > y$ and $\sigma = \{x \mapsto x'\}$, $F\sigma = x' + 1 > y$.

Proof. Proof by contradiction. The verification conditions for an assertion in s', obtained by symbolic execution of $s \cdot s'$ and $v \cdot s'$, can be expressed as:

$$\mathsf{ST}(s)\sigma \wedge F, \qquad \mathsf{ST}(v)\sigma \wedge F$$

where σ and F are a variable substitution and a new constraint, both of which are introduced during the symbolic execution of s'. Our goal is to prove that if $\mathsf{ST}(s)\sigma \wedge F$ is satisfiable, then so is $\mathsf{ST}(v)\sigma \wedge F$:

$$\exists m_1.m_1 \models \mathsf{ST}(s)\sigma \wedge F \implies \exists m_2.m_2 \models \mathsf{ST}(v)\sigma \wedge F \tag{10}$$

Suppose (10) does not hold:

$$\exists m_1.m_1 \models \mathsf{ST}(s)\sigma \wedge F, \qquad \forall m_2.m_2 \models \neg\mathsf{ST}(v)\sigma \vee \neg F \tag{11}$$

From (11), we have

$$\text{(a) } \exists m_1.m_1 \models \mathsf{ST}(s)\sigma, \qquad \text{and} \qquad \text{(b) } \exists m_1.m_1 \models F \tag{12}$$

and

$$\text{(c) } \forall m_2.m_2 \not\models \mathsf{ST}(v)\sigma, \qquad \text{or} \qquad \text{(d) } \forall m_2.m_2 \not\models F \tag{13}$$

In the case (d) of (13), we close the branch, since (b) and (d) are contradictory. Now we consider the case (c). Given a variable assignment m, let recover be the function that returns a variable assignment where all primed global variables in $\mathsf{Ran}(\sigma)$ are replaced with their original (unprimed) global variable names:

$$\mathsf{recover}(m) = \{\mathsf{org}(x) \mapsto v \mid (x \mapsto v) \in m\}$$

where $\mathsf{org}(x) = \textbf{if } (y \mapsto x) \in \sigma \wedge y \in G \textbf{ then } y \textbf{ else } x$. Note that: (i) the state conditions for s and v, $\mathsf{ST}(s)$ and $\mathsf{ST}(v)$, can be obtained from $\mathsf{ST}(s)\sigma$ and $\mathsf{ST}(v)\sigma$ by replacing every primed global variable $x' \in \mathsf{Ran}(\sigma)$ with its original variable $x \in G$. Further note that: (ii) the evaluation of x' under m and x under $\mathsf{recover}(m)$ are the same, that is, $\forall x, x'.m(x') = (\mathsf{recover}(m))(x)$ where $\sigma(x) = x'$. From (a), (c), (i), and (ii), we have

$$\exists m'_1.m'_1 \models \mathsf{ST}(s), \qquad \forall m'_2.m'_2 \not\models \mathsf{ST}(v) \tag{14}$$

where $m'_1 = \mathsf{recover}(m_1)$ and $m'_2 = \mathsf{recover}(m_2)$. From (14), for some variable assignment k such that $k \models \mathsf{ST}(s)$ and $k \not\models \mathsf{ST}(v)$, we have

$$k \models \exists \mathbf{X}.\mathsf{ST}(s), \qquad k \not\models \exists \mathbf{Y}.\mathsf{ST}(v) \tag{15}$$

where $\mathbf{X} = \mathsf{FV}(\mathsf{ST}(s)) \setminus G$ and $\mathbf{Y} = \mathsf{FV}(\mathsf{ST}(v)) \setminus G$. From (15), we have

$$k \not\models \exists \mathbf{X}.\mathsf{ST}(s) \to \exists \mathbf{Y}.\mathsf{ST}(v) \tag{16}$$

According to the assumption (i.e., $\mathsf{subsumed}(s, v)$) of Theorem 3.3, we have

$$\forall n.n \models \exists \mathbf{X}.\mathsf{ST}(s) \to \exists \mathbf{Y}.\mathsf{ST}(v) \tag{17}$$

(16) and (17) are contradictory. All branches, (c) and (d), are closed. □

As a side remark, when we prune s' because subsumed(s', v) holds for some validated sequence $v \in V$, it is possible that extensions of v may later be pruned as well, due to being subsumed by another validated sequence $v' \in V \setminus \{v\}$ in future iterations. Even in such cases, our technique still guarantees to safely reduce the search space without compromising vulnerability detection. This is because such v' can appear only after its immediate extensions have been added to W in earlier iterations, as ensured by lines 8, 9, and 20 in Algorithm 1.

4 Optimizations

4.1 Constraint Simplification

SMT solving is a well-known performance bottleneck of SMT-based analyzers [25]. To mitigate this issue, we adopt existing constraint simplification techniques, such as constant folding, semantics-preserving rewriting (e.g., replacing $e + 0$ with e), and property-focused simplification [65]. Integrating these into SmarTrim is mostly straightforward, but eliminating *isolated* variables (EIV, an instance of Z3's tactic `elim-uncnstr` [18, 19]) requires special care. Specifically, for the following two cases, we retain constraints even if they contain isolated variables.

First, we keep isolated function-input-parameters in verification conditions. For example, consider a verification condition $F : y_i^e = a \wedge F_2 \wedge \cdots \wedge F_n$, where y_i^e denotes the input parameter y at the entry point (e) of a transaction whose identifier is i. Suppose that y_i^e is isolated in that it does not appear in $F_2 \wedge \cdots \wedge F_n$. Then, we can simplify F to an equisatisfiable formula $F' : F_2 \wedge \cdots \wedge F_n$, but we retain the original formula F. This is because SmarTrim aims to provide vulnerable sequences with concrete argument values per transaction, rather than simply detecting vulnerabilities.

Second, we keep isolated global variables in subsumption conditions (Definition 3.2), because every global variable is needed to precisely reason about reachable global states.

4.2 Reducing Pruning Overhead

Even with constraint simplifications, verifying subsumption conditions in first-order logic (Definition 3.2) still poses a substantial performance issue: they involve quantified formulas, which are harder for SMT solvers to process in general, and furthermore, for every new sequence, we validate its redundancy against previously validated sequences (line 13 in Algorithm 1). We describe methods to reduce this pruning cost.

4.2.1 **Validity Templates**. We use the three validity templates to check subsumption conditions efficiently, without invoking off-the-shelf SMT solvers. Let Def (t) be the set of global variables defined (via assignments) in t, and Use (t) be the set of global variables used in t.

NoDef. Let t be a transaction such that Def $(t) = \emptyset$. Then, for any non-empty transaction sequence s, s subsumes $s \cdot t$, as t cannot extend global states reachable from s. This idea is formalized by the below inference rule, whose soundness proof is in the supplement B.

$$\frac{\text{Def}\,(t) = \emptyset}{\text{subsumed}\,(s \cdot t, s)} \quad (\text{NoDef})$$

Inter. From Example 3 in Section 2.1, recall that the order between two transactions (t_1 and t_2) does not affect the reachable global states, if the global variables defined in t_1 do not appear in t_2, and vice versa. We designed Inter based on this observation.

$$\frac{Y_1 : \text{Def}\,(t_1) \cap (\text{Def}\,(t_2) \cup \text{Use}\,(t_2)) = \emptyset, \quad Y_2 : \text{Def}\,(t_2) \cap (\text{Def}\,(t_1) \cup \text{Use}\,(t_1)) = \emptyset}{\text{subsumed}\,(s \cdot t_2 \cdot t_1, s \cdot t_1 \cdot t_2)} \quad (\text{Inter})$$

We provide the intuition in more detail. Y_1 ensures that global variables defined in t_1 are neither redefined nor constrained by t_2. Likewise, Y_2 guarantees that global variables defined in t_2 are

neither redefined nor constrained by t_1. From Y_1 and Y_2, it follows that, for any global state, the portion extended by one transaction is unaffected by the other. Thus, the execution order between t_1 and t_2 does not alter the final global states. The soundness proof is in the supplement C.

Proof with Previous Subsumptions. Below is a template that leverages previously identified subsumption relations. Recall the contract in Figure 1. Suppose $\mathsf{subsumed}(t_0 \cdot t_1, t_0)$ holds where $t_1 = \texttt{burnFrom(C, n)}$. Suppose further $t_0 \cdot t_2$ has been analyzed, where $t_2 = \texttt{setOwner(a)}$. Then, using Prev, we can efficiently detect that $t_0 \cdot t_2$ subsumes a new sequence $t_0 \cdot t_2 \cdot t_1$. See the supplement D for the soundness proof.

$$\frac{\begin{array}{c} Y_1 : \mathsf{subsumed}\ (s \cdot t_1, s) \\ Y_2 : \mathsf{Def}\ (t_1) \cap (\mathsf{Def}\ (t_2) \cup \mathsf{Use}\ (t_2)) = \emptyset, \quad Y_3 : \mathsf{Def}\ (t_2) \cap (\mathsf{Def}\ (t_1) \cup \mathsf{Use}\ (t_1)) = \emptyset \end{array}}{\mathsf{subsumed}\ (s \cdot t_2 \cdot t_1, s \cdot t_2)}\ (\textsc{Prev})$$

4.2.2 **Lightweight Condition**. After applying the validity templates (Section 4.2.1) and before checking subsumption conditions using an SMT solver, we check an alternative lightweight condition, which we found is often more efficiently processed by the Z3 SMT solver [32]. Given a sequence $s = s' \cdot t_n$ where $s' = t_1 \cdots t_{n-1}$, if t_n is a state-preserving transaction that does not modify the global state (e.g., Example 1 in Section 2.1), s is redundant with s':

$$\frac{F : \mathsf{ST}(s' \cdot t_n) \rightarrow \bigwedge_{g \in G} g_n^e = g \text{ is valid}}{\mathsf{subsumed}\ (s' \cdot t_n, s')}\ (\textsc{NoModify})$$

where g_n^e is a variable that denotes g at the entry point of t_n (Section 3.1). Note that the hypothesis ensures $A(s' \cdot t_n) = A(s')$, from which we can derive the conclusion.

4.2.3 **Selective Pruning**. We use heuristics to invoke an SMT solver only when pruning is likely to succeed (i.e., subsumption conditions are likely to hold). Given a candidate sequence s and a previous sequence $v \in V$, we directly check $\mathsf{subsumed}(s, v)$ with an SMT solver only if: (i) the cumulative time spent on direct SMT-based subsumption checking does not exceed 30% of the total SMT solving time, and (ii) one of the following three conditions holds.

(1) The length of s is no larger than three (excluding an initial transaction t_0), and v is a prefix of s. That is, $s = v \cdots t_n$ where $n \leq 3$, and $v = t_0$ or $v = t_0 \cdots t_i$ $(0 < i < n)$.
(2) $s = t_0 \cdot t_1$ and $v = t_0 \cdot t_2$, where t_1 and t_2 are single transactions, are expected to update the same set of global variables. That is, $\mathsf{Def}\ (t_1) = \mathsf{Def}\ (t_2)$.
(3) s and v, whose lengths are two, share the two transactions but in different orders. That is, $s = t_0 \cdot t_1 \cdot t_2$ and $v = t_0 \cdot t_2 \cdot t_1$.

These three conditions are devised based on the observation that: s is likely to be subsumed by v when they share syntactically or semantically similar features.

Note that our selective pruning preserves the pruning safety in Theorem 3.3. If the conditions (i) and (ii) hold, we directly verify the subsumption condition ($\mathsf{subsumed}\ (s, v)$) using an SMT solver and perform pruning only when the subsumption condition holds; thus, the theorem applies. Otherwise, we conservatively regard the subsumption condition as *false* and skip pruning; therefore, the safety is trivially preserved.

5 Implementation

We implemented SmarTrim in OCaml on top of SmarTest [65], an open-source symbolic execution tool for Solidity smart contracts. In particular, we modified SmarTest's basic sequence-generation mechanism so that infeasible transaction sequences are detected and pruned lazily rather than

eagerly (Section 3.1). We use the Z3 SMT solver [23, 32] (v.4.15.2) to check the satisfiability of path conditions and verification conditions, and the validity of subsumption conditions.

Preprocessing and Path Enumeration Details. For testing efficiency, our implementation of Algorithm 1 uses two heuristics. First, during preprocessing (before line 1), function call statements are inlined up to a maximum call depth of 3, and loops are unrolled twice. As a result, paths that require deeper call depths or additional loop iterations are excluded from symbolic execution. Second, during path enumeration for each function (line 1), we terminate the enumeration once the number of collected paths exceeds 50. Since our implementation enumerates function paths in breadth-first search (BFS) order and checks this stopping condition at the end of each BFS iteration, the final number of collected paths per function may exceed 50.

Vulnerability Oracles. SmarTrim currently detects 7 kinds of vulnerabilities. We reused vulnerability detection rules from SmarTest [65] for 6 types: integer over/underflow, division-by-zero, assertion violation, ERC20 standard [8] violation, Ether-leak, and suicidal vulnerability. In addition, we implemented a new test oracle to detect reentrancy vulnerabilities by checking whether attackers can steal Ethers through internal transactions (i.e., via external function calls).

Concrete Validator. To automatically validate whether vulnerable transaction sequences generated by SmarTrim can indeed reproduce the detected vulnerabilities in real executions, we implemented a concrete validator based on the Foundry toolkit [20]. Specifically, our validator replays generated sequences in a local environment and checks for violations of the associated safety conditions. Note that the validator is not part of SmarTrim's analysis pipeline; rather, it is used only as a post hoc evaluation tool to check whether the reported alarms can be reproduced in real executions.

Solver Timeout. We set the Z3 solver's timeouts to 1.5 seconds for each feasibility check (line 11 in Algorithm 1) and subsumption-condition check (line 13), and to 90 seconds for each verification-condition check (line 18). We allocated more time for verification condition checking to avoid missing opportunities for vulnerability detection.

6 Evaluation

We performed experiments to answer the research questions below:

- **RQ1**: How effective is SmarTrim at detecting vulnerabilities compared to state-of-the-art analyzers? (Section 6.1)
- **RQ2**: How significant is our pruning for the performance of SmarTrim? (Section 6.2)
- **RQ3**: Can SmarTrim discover new bugs in recent real-world smart contracts? (Section 6.3)

6.1 Comparison with Existing Analyzers

6.1.1 **Setup**. We describe the setup for comparative experiments.

Target Vulnerabilities. We focused on evaluating the ability to detect four kinds of security-critical vulnerabilities that have been extensively studied in prior work: integer over/underflow (IO), Ether-leak (EL; Ether-stealing without reentrancy), suicidal (SU; insecure access to `selfdestruct` or `suicide` in Solidity), and reentrancy (RE; Ether-stealing via reentrancy from external contracts).

We note that, despite the Cancun hard fork where `selfdestruct` (or `suicide`) no longer removes contract code from the blockchain as per EIP-6780 [26], we decided to include SU as a target vulnerability in our experiments. This is because the two instructions (`selfdestruct`, `suicide`) remain security-sensitive instructions that should be accessible only to privileged users: insecure accesses to them can trigger unexpected Ether balance transfers, leading to denial-of-service in attacked contracts.

Benchmark. Drawing on a range of existing studies [2, 28, 31, 53, 65, 67, 86], we built three datasets, comprising a total of 814 Solidity smart contracts with known and annotated bugs.

- LS Dataset: 491 contracts containing EL or SU: 84 and 407 from [65] and [53, 86], respectively.
- IO Dataset: 274 CVE-listed contracts containing IO bugs from [2, 65] after deduplication.
- RE Dataset: 49 contracts with RE: 43, 3, and 3 selected from [67], [31], and [28] respectively.

For each dataset, the average number of lines per contract is: 411 (LS Dataset), 235 (IO Dataset), and 226 (RE Dataset). The above contract counts show the statistics after preprocessing, such as deduplication or removal of contracts without in-scope vulnerabilities. For example, to create LS Dataset, we selected 407 contracts, out of 911 contracts reported as vulnerable by SPCon [53] and PrettySmart [86]. This is because the two analyzers mainly target access-control bugs (i.e., insecure access to certain variables or instructions), which do not always align with EL or SU. We manually identified 504 contracts that contain neither EL nor SU, and excluded them from LS Dataset. Additional details on preprocessing are described in the supplement E.

Ground Truths. For 274 contracts in IO Dataset, the CVE-reported vulnerabilities are the ground-truth bugs. For 84 contracts [65] in LS Dataset and 43 contracts [67] in RE Dataset, we reused the ground truths from related prior work. For the remaining contracts, we established ground truths through careful manual inspection, which involved iteratively refining them by manually analyzing each tool's detection results and adding any new bugs that were not included in the existing ground truths. Thus, for EL, SU, and RE, we deem that a tool has generated a false positive if it raises an alarm for a location outside the ground truths.

Competing Tools. To thoroughly assess the practicality of SmarTrim, we selected 11 recent analyzers based on different analysis approaches:

- Symbolic execution: SmarTest [65], Mythril [11], LENT-SSE [84], AChecker [37], Sailfish [28], SliSE [76]
- Fuzzing: Smartian [31], ConFuzzius [73], RLF [71], EF/CF [62]
- Static analysis: Slither [33]

Among the six symbolic executors, the first three [11, 65, 84] can produce vulnerable transaction sequences like SmarTrim, and the others are bug-finders [28, 37, 76] that use symbolic execution to reduce false positives. ConFuzzius [73] is a hybrid fuzzer that leverages symbolic execution to effectively explore complicated paths. We used the latest (as of August 2025) versions of each tool from public GitHub repositories (SmarTest [17], LENT-SSE [10], Smartian [15], AChecker [3], Slither-v0.11.3) and public Docker images (Mythril-v0.24.8, RLF [13], Sailfish [14], SliSE [7], ConFuzzius [5], EF/CF [6]).

Although we curated benchmarks from SPCon [53] and PrettySmart [86], we did not include them in our comparative experiments. SPCon's free usage was limited due to its reliance on the `Bitquery` [4] API. For PrettySmart, we could not be confident that we were using it correctly, as we failed to reproduce the experimental results in our environment using its artifact [12].

Tool Execution Details. We use the default options for all competing tools, with one exception: for LENT-SSE [84], we set the Z3 SMT solver timeout to 5 seconds, and limited the maximum transaction sequence length to four (excluding initial transactions), as these configurations yielded the best performance in its original study [84]. Since RLF [71] raised runtime errors for contracts with parameterized constructors when no arguments were provided, we supplied random arguments to prevent such errors. As SmarTrim's default search strategy (line 7 in Algorithm 1), we adopted an increasing-order approach that prioritizes shorter candidate sequences. For each dataset, we executed the tools with only the relevant checkers enabled, except for Smartian and ConFuzzius that do not support such options.

Table 1. The detection results of each tool on the ground-truth vulnerabilities. Line: the number of vulnerable lines detected by each tool. Func: the number of vulnerable functions detected by each tool. All (n): the results on all contracts. Com (n): the results on the contracts that were commonly analyzed by all tools without runtime errors or timeouts (n shows the number of such contracts). n/a: the tool neither covers the vulnerability nor provides line-level results.

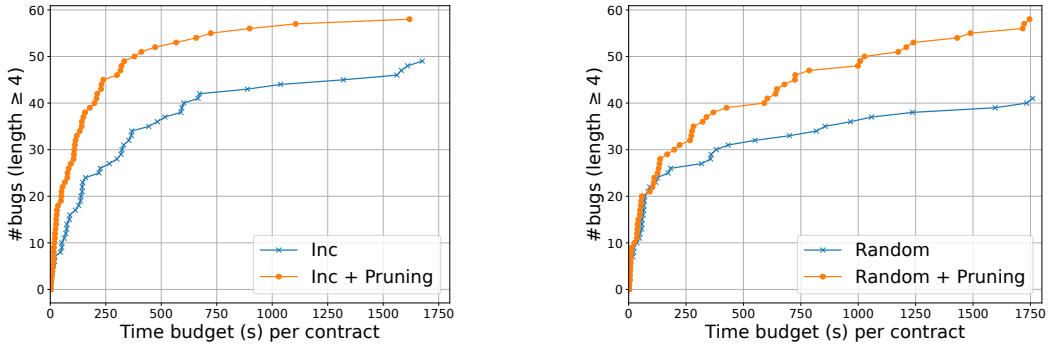
Tool Name	LS Dataset								IO Dataset				RE Dataset				Total	
	EL				SU				IO				RE					
	All (491)		Com (254)		All (491)		Com (254)		All (274)		Com (217)		All (49)		Com (21)		All (814)	
	Func	Line	Func	Line	Func	Line	Func	Line	Func	Line	Func	Line	Func	Line	Func	Line	Func	Line
SmarTrim	**364**	**373**	**225**	**232**	**114**	**114**	**62**	**62**	**136**	**154**	**115**	**122**	34	**35**	**21**	**21**	**648**	**676**
SmarTest [65]	304	313	202	208	101	102	59	59	132	146	114	118	n/a	n/a	n/a	n/a	537	561
EF/CF [62]	322	n/a	190	n/a	108	n/a	59	n/a	n/a	n/a	n/a	n/a	21	n/a	12	n/a	451	n/a
Smartian [31]	154	n/a	112	n/a	71	n/a	40	n/a	133	n/a	112	n/a	22	n/a	18	n/a	380	n/a
ConFuzzius [73]	n/a	108	n/a	74	n/a	88	n/a	56	n/a	99	n/a	95	n/a	30	n/a	18	n/a	325
RLF [71]	210	n/a	148	n/a	84	n/a	54	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	294	n/a
Mythril [11]	85	88	54	57	95	95	55	55	57	61	49	53	28	n/a	20	n/a	265	244
AChecker [37]	173	n/a	110	n/a	79	n/a	43	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	252	n/a
Slither [33]	73	83	40	45	59	n/a	33	n/a	n/a	n/a	n/a	n/a	**42**	n/a	19	n/a	174	83
LENT-SSE [84]	18	18	12	12	67	67	50	50	39	38	36	35	22	n/a	16	n/a	146	123
SliSE [76]	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	33	n/a	21	n/a	33	n/a
Sailfish [28]	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	27	n/a	20	n/a	27	n/a

For machine learning-based analyzers (SmarTest [65], RLF [71]), we used the pre-trained models. Specifically, SmarTest employed 4-fold cross-validation in its original experiments [65]. To evaluate it under favorable conditions, we followed the same 4-fold cross-validation setup for the contracts overlapping with SmarTest's original dataset (i.e., applied fold-specific models to each fold). For new contracts [53, 86] in LS Dataset, we used the model trained on SmarTest's entire contracts with EL and SU vulnerabilities.

Hardware and Testing Resources. The evaluation was conducted on Ubuntu machine with AMD Ryzen Threadripper 3970X CPU (3.7 GHz) (32 cores and 64 threads in total, 62 GB of memory). We ran each analyzer with at most 24 threads, which was sufficient for most tools to run without out-of-memory errors. By exception, RLF [71] was limited to 3 threads, as it frequently raised out of memory errors under the default setting. For each tool, we allocated a 30-minute internal timeout and a 35-minute external timeout per contract, except for LENT-SSE [84], ConFuzzius [73] and EF/CF [62]. For LENT-SSE, because its implementation does not support an internal timeout option and often failed to produce analysis reports within 35 minutes, we set its external timeout to 180 minutes. For ConFuzzius and EF/CF, since they failed to produce analysis results on several benchmarks within the 35-minute external timeouts despite using the internal timeout options, we set their external timeouts to 40 and 50 minutes, respectively.

6.1.2 **Results**. Table 1 compares the vulnerability-detection results among the analyzers. The column All shows the results for the entire contracts of each dataset. For a more balanced comparison, the column Com shows the results for the contracts that were successfully analyzed by all 12 tools (i.e., without runtime failures and timeouts). The column Line presents the number of line-level vulnerabilities detected by each tool. For the tools that do not provide line-level detection results, we provide the function-level results only in the column Func.

The results show that SmarTrim outperforms the 11 competing analyzers in vulnerability detection. In total, SmarTrim discovered 81.8% ($\frac{373}{456}$), 93.4% ($\frac{114}{122}$), 86.0% ($\frac{154}{179}$), and 67.3% ($\frac{35}{52}$) of



(a) Comparison under increasing-order search

(b) Comparison under randomized search

Fig. 3. SmarTrim with vs. without pruning.

the known EL, SU, IO, and RE line-level bugs in the ground truths. By contrast, SmarTest, the second-best tool, detected 68.6% ($\frac{313}{456}$), 83.6% ($\frac{102}{122}$), and 81.6% ($\frac{146}{179}$) of the known EL, SU, and IO bugs. Although Slither is more effective than SmarTrim in detecting RE, SmarTrim remains highly competitive: SmarTrim provides vulnerable transaction sequences unlike Slither, and found more RE vulnerabilities than the other remaining tools.

In total, SmarTrim detected 1,812 potential bugs (including the ones beyond the ground truths) along with their corresponding vulnerable transaction sequences. Using our concrete validator (Section 5), we confirmed that 92.8% of these 1,812 vulnerable sequences can successfully replay the detected safety violations. The remaining 7.2% failed to reproduce the alarms due to, for example, imprecise handling of cryptographic hash functions (e.g., `keccak256`) and SmarTrim's lack of support for certain Solidity features.

Pruning Cost. The pruning overhead, including SMT solving of the original subsumption-condition and pruning-related optimizations (Section 4.2), was modest. On average, our pruning took 197.7 seconds per contract, accounting for 18.1% of the total analysis time.

Analysis Time. The following shows the total runtime of each tool for LS Dataset, IO Dataset, and RE Dataset: SmarTrim (5h 8m, 6h 4m, 30m), SmarTest (5h 42m, 6h 2m, n/a), EF/CF (11h 12m, n/a, 1h 12m), Smartian (10h 0m, 6h 0m, 1h 0m), ConFuzzius (8h 30m, 4h 51m, 1h 0m), RLF (2d 13h 52m, n/a, n/a), Mythril (9h 10m, 6h 4m, 1h 1m), AChecker (1h 2m, n/a, n/a), Slither (15s, n/a, 7s), LENT-SSE (22h 13m, 16h 48m, 4h 21m), SliSE (n/a, n/a, 35m), Sailfish (n/a, n/a, 1m 40s).

6.2 Significance of Our Pruning

Setup. We conducted an ablation study to evaluate the efficacy of our pruning technique (Section 3.2). To this end, we considered four variants of SmarTrim. In Figure 3, Inc and Inc+Pruning denote SmarTrim that performs an increasing-order search (Section 6.1.1) without and with pruning, respectively: Inc+Pruning corresponds to SmarTrim in Table 1. To assess the general applicability of our pruning, we also built two additional variants: Random and Random+Pruning. Random indicates SmarTrim that performs a randomized search without pruning: at each iteration of the repeat-until loop in Algorithm 1, it randomly selects a candidate from the workset (line 7) and generates its successor sequences (line 21) up to length 5 (excluding initial transactions). Random+Pruning denotes SmarTrim with the same randomized search and pruning.

To evaluate each variant with and without pruning, we created the cactus plots in Figure 3. We focused on comparing their performance in finding "difficult-to-detect" (hereafter, shortly

"elusive") vulnerabilities, which require transaction sequences with lengths at least 4 (excluding initial transactions) for detection. In Figure 3(a) and (b), the x-axis represents the analysis time per contract, and the y-axis shows the cumulative number of detected elusive vulnerabilities over time across all three datasets. More specifically, for each variant, we record the timestamp at which each elusive vulnerability is detected, sort these timestamps across all 814 contracts, and compute the cumulative number of detected elusive bugs over time. To compute the y-axis values, we established the ground-truth for elusive vulnerabilities by manually inspecting the bug-detection results of each variant, excluding those discovered with sequences shorter than 4.

Results. Figure 3 demonstrates the importance of our pruning in enhancing vulnerability-detection of SmarTrim. While Inc and Random generated 50 and 42 vulnerable transaction sequences of lengths ≥ 4, Inc+Pruning and Random+Pruning produced 59 and 59 such sequences, respectively. The total vulnerabilities detected by each variant are: Inc (1802) vs. Inc+Pruning (1812), and Random (1788) vs. Random+Pruning (1802). In addition, we measured statement coverage during the symbolic explorations (line 16 of Algorithm 1): Inc (97.70%), Inc+Pruning (97.72%), Random (97.68%), and Random+Pruning (97.69%).

Validation of Pruning Safety. We empirically confirmed the safety of our pruning (Section 3.2). In our ablation study, 55 bugs were detected only by the variants without pruning (5 from Inc, 50 from Random). However, we found that these bugs were missed by the pruning variants due to the nondeterminism of the Z3 solver, not because the pruning itself is unsafe: when we increased the Z3 solver's timeout to 10 minutes, the variants with pruning successfully detected all 55 bugs.

Impact of Optimization Techniques. We also found that the optimization techniques in Section 4.2 help reduce pruning overhead. SmarTrim without the optimizations in Section 4.2 detected 1,750 bugs (23 elusive), whereas the full SmarTrim detected 1,812 bugs (59 elusive).

More detailed statistics show that the optimizations improve vulnerability-finding ability by reducing expensive subsumption checks and enabling the exploration of more transaction sequences. On average, the full SmarTrim explored 20,478 sequences per contract, performed 104 direct subsumption-condition checks using an SMT solver, and identified 584 sequences as redundant overall and pruned their extensions. By contrast, SmarTrim without the optimizations, on average, explored 2,521 sequences, performed 6,626 direct subsumption-condition checks, and identified 33 sequences as redundant and pruned their extensions.

6.3 Finding Vulnerabilities in the Wild

We also confirmed the applicability of SmarTrim to recently deployed, real-world smart contracts. We collected 4,466 Solidity smart contracts from Etherscan [9] in February 2025, and after deduplication (Section 6.1), selected 2,504 as validation targets. We ran SmarTrim on the 2,504 contracts to detect IO, EL, SU, RE bugs, as well as ERC20 specification violations. After carefully triaging the alarms reported by SmarTrim, we identified 5 critical IO bugs and 7 critical ERC20 specification violations across 10 contracts, excluding benign cases (e.g., benign alarms arising from not accounting for contract-specific specifications, vulnerabilities in small and presumably educational contracts). These results indicate that SmarTrim is effective at finding vulnerabilities in real-world contracts. Since the contact information of the developers of the 10 contracts is not available, we plan to report our findings to the CVE teams.

More interestingly, through our study, we learned that detecting IO bugs remains important even in recent ($\geq$ v0.8.0) Solidity contracts. Since Solidity v0.8.0 (December 2020), integer over/underflows revert transactions by default, unless arithmetic operations are performed within `unchecked` blocks. Thus, it might be reasonable to expect IO bugs to be rare in recent contracts. However, 790 of the

```
1  pragma solidity ^0.8; // compatible with Solidity 0.8.x (with default checks enabled)
2  contract VulnerableRedeem {
3    constructor() { owner = msg.sender; balances[owner] = 1e15;  totalSupply = 1e15; }
4
5    function transfer(address to, uint value) public {
6      balances[msg.sender] -= value;  // underflow-safe due to default checks
7      balances[to] += value;          // overflow-safe due to default checks
8    }
9
10   function redeem(uint amount) public {
11     require(msg.sender == owner);
12     require(totalSupply >= amount); // fix: require(balances[owner] >= amount);
13     unchecked {                     // default safety checks are disabled
14       balances[owner] -= amount;    // integer underflow bug
15       totalSupply -= amount;
16     }
17   }
18 }
```

Fig. 4. A simplified real-world contract with an integer underflow bug detected by SmarTrim.

2,504 target contracts use unchecked blocks to reduce gas fees resulting from default arithmetic-safety checks, while relying on developers' custom guards. By exploiting these potential risks of arithmetic errors, SmarTrim uncovered 16 true IO bugs, of which 5 are critical. The main cause was the use of unchecked blocks guarded by improper checks that admit corner cases.

For example, SmarTrim found the critical integer underflow bug at line 14 in Figure 4. Despite the guard at line 12, the following sequence triggers the bug: (1) `transfer (A,1)` where `msg.sender = owner` and `A` $\neq$ `owner`, (2) `redeem (1e15)` where `msg.sender = owner`. The underflow at line 14 inflates the owner's balance to a large value ($2^{256} - 1$) and potentially lowers the token price. To fix this issue, line 12 should be modified as: `require(balances[owner] >= amount);`

In addition, our investigation further highlights the usefulness of SmarTrim over the competing analyzers. Of the 5 critical IO bugs detected by SmarTrim, each of the five tools that support IO detection–SmarTest, Smartian, ConFuzzius, Mythril, and LENT-SSE–detected at most three, and all five tools failed to detect one.

7 Limitations and Future Work

We outline the current limitations of SmarTrim, which we leave as directions for future work.

Applicability of Pruning. We demonstrated the effectiveness of our pruning technique under two simple search strategies (Section 6.2). However, its effectiveness when combined with other search strategies remains to be investigated. In addition, we showed its usefulness for four kinds of vulnerabilities (Sections 6.1 and 6.2), but the underlying idea is not limited to them and could potentially be extended to other bug types, including those that enable profit-accumulating attacks [39]. We leave such extensions for future work.

False Negatives. In the comparative experiments (Section 6.1), SmarTrim failed to detect 131 ground-truth bugs for three main reasons. First, to reduce false positives, SmarTrim filters out transactions containing external function-call statements that invoke functions from other contracts and return values; this caused 29 false negatives. Second, transaction sequences necessary to trigger bugs have not been analyzed within a time budget, indicating that SmarTrim's performance still needs improvement; this accounted for 29 false negatives. Third, when collecting function paths (line 1 in Algorithm 1), SmarTrim discards paths in which call depths exceed a predetermined bound (i.e., paths that still contain function-call statements after inlining up to a predefined depth), thereby missing transactions necessary for bug detection; this lead to 23 false negatives. Other

examples include: the inability to detect bugs inside the bodies of invoked external functions, limited support for hash functions, solver timeouts, and out-of-memory errors.

False Positives. For IO, SmarTrim produced 8 false positives (i.e., cases where transaction sequences that could violate the safety conditions do not exist), due to implementation errors in its frontend. We fixed these errors in our maintained implementation.

For EL, SU, and RE, SmarTrim generated 46 false positives (i.e., alarms beyond our ground truths; Section 6.1.1) in total. The main cause was the limitations of the vulnerability oracles (Section 5), which do not consider contract-specific specifications. For instance, sending Ethers to unprivileged users who have never invested in the contract can be legal in the context of gambling contracts; we provide an example in the supplement F. The following shows the precision ($\frac{\#TP}{\#TP+\#FP}$) of the analyzers for EL, SU, and RE: SmarTrim (91.4%), SmarTest (94.1%), EF/CF (85.1%), Smartian (96.5%), ConFuzzius (91.9%), RLF (91.3%), Mythril (69.6%), AChecker (77.2%), Slither (59.2%), LENTSSE (94.4%), SliSE (80.5%), and Sailfish (72.9%).

Accelerating Subsumption Checking. Developing additional techniques to reduce pruning overhead (i.e., to optimize subsumption-condition checks) would further enhance the practicality of SmarTrim. Existing analysis methods, however, would not be readily applicable to our purpose and would require nontrivial adjustments. For example, suppose we introduce a new symbolic transaction t_5 : `setXNot10(m)` by adding the function `function setXNot10 (uint y) { require(y != 10); x = y; }` to the contract in Figure 2. In this case, $p : t_0 \cdot t_3$ is not subsumed by $r : t_0 \cdot t_5$, because x after executing p is 10 but x after executing r cannot be 10. Yet a conventional (non-disjunctive) interval analysis might incorrectly conclude that p is subsumed by r, leading to unsafe pruning of subsequent sequences of r. This is because the resulting interval values for x after analyzing p and r would be $[10, 10]$ and $[0, 2^{256} - 1]$ respectively, and the latter subsumes the former in the interval domain.

8 Related Work

Symbolic Execution of Smart Contracts. SmarTrim complements existing approaches [28, 34, 47, 54, 55, 65, 76, 81–85] that aim to speed up symbolic execution of smart contracts, by introducing a unique contribution: the ability to effectively prune redundant transaction sequences based on reachable global states. For example, while our technique can prune more redundant paths (e.g., Examples 1 and 2 in Section 2.1) compared to data dependency-based pruning methods [81, 82], we may adopt such pruning as an alternative when subsumption checking incurs too much overhead and becomes practically infeasible (e.g., sequences longer than length three, see Section 4.2.3). Moreover, once paths irrelevant to targeted vulnerabilities are eliminated using the methods in [55, 76, 83], SmarTrim can further prune vulnerability-relevant but redundant paths. EthRacer [47] focuses specifically on pruning redundant sequences arising from invalid transaction orderings that trigger runtime exceptions. By contrast, SmarTrim can eliminate redundant sequences even when they contain valid transaction orderings. For instance, in Example 3 of Section 2.1, EthRacer cannot identify that p is redundant with q because both $t_1 \cdot t_2$ and $t_2 \cdot t_1$ are valid orderings that do not trigger runtime exceptions. Our pruning technique can be integrated with other orthogonal optimization approaches [28, 34, 54, 65, 84, 85] to further enhance the performance of symbolic execution.

Symbolic Execution of Conventional Programs. Our work is also differentiated from existing symbolic execution approaches [24, 27, 38, 42–45, 48, 50, 56, 61, 74, 79, 80] for pruning redundant paths in traditional programs, such as C or Java. In particular, we present a new pruning technique specifically designed to analyze smart contracts, that is, subsumption checking based on reachable global states. We discuss prior studies in more detail.

The approaches in [42–45, 56, 79, 80] operate in two main steps. First, for each branch point, they generate a constraint that summarizes the behavior of explored path suffixes, using interpolation [42–45, 56] or weakest precondition generation [79, 80]. Then, when exploring new paths with the same suffixes, they backtrack to the most recent branches if the current state constraints imply the summarized constraints, thereby avoiding redundant exploration of path suffixes that do not lead to new states. This idea is insufficient for our purpose, as we also aim to eliminate paths with different transaction suffixes (e.g., Examples 1-3 in Section 2.1).

Anand et al. [24] presented a subsumption checking method specialized for heap structures in Java programs. Guo et al. [38] proposed a technique for eliminating redundant execution paths in multithreaded C/C++ programs. Several researchers [27, 48, 50, 61, 74] developed methods to discard paths functionally equivalent to others, whereas we aim to prune subsumed paths as well.

9 Conclusion

Automatically detecting vulnerabilities in smart contracts using symbolic execution remains challenging, because the number of transaction sequences to explore is intractably large. To address this limitation of existing approaches, we proposed SmarTrim, a novel pruning technique to enhance the performance of symbolic execution for smart contracts. The central idea is to automatically detect and eliminate transaction sequences that are redundant with previously analyzed sequences, based on reachable global states. We formally presented our technique and validated its effectiveness through extensive comparative experiments involving 11 state-of-the-art analyzers.

10 Data Availability

Our replication artifact, including SmarTrim's source code and our datasets, is publicly available on both Zenodo [68] and GitHub [69].

Acknowledgments

This work was supported by a Korea University Grant.

References

[1] [n. d.]. 1inch resolver suffers $5M hack due to smart contract vulnerability. https://www.tradingview.com/news/cointelegraph:b5d53305d094b:0-1inch-resolver-suffers-5m-hack-due-to-smart-contract-vulnerability/. Accessed: April 2026.
[2] [n. d.]. 487 smart contracts that contain arithmetic vulnerabilities with assigned CVE IDs. https://github.com/kupl/VeriSmart-benchmarks/tree/master/benchmarks/cve. Accessed: April 2026.
[3] [n. d.]. Achecker (e4e47efb7f39bc15425fca5568528386abe3cadc). https://github.com/DependableSystemsLab/AChecker. Accessed: April 2026.
[4] [n. d.]. Bitquery. https://bitquery.io/. Accessed: April 2026.
[5] [n. d.]. ConFuzzius (4e7e1e71358221774d2ac93ec34d590d052ad608). https://github.com/christoftorres/ConFuzzius. Accessed: April 2026.
[6] [n. d.]. EF/CF - the Extremely Fast (ethereum smart) Contract Fuzzer (1a7b2da6e4b5e32ee54fd8f09f3de57c811effc9). https://github.com/uni-due-syssec/efcf-framework. Accessed: April 2026.
[7] [n. d.]. Efficient Detection of Reentrancy Vulnerabilities in Complex Smart Contracts (02832b17f385b5e9a906e4d892ea65401846566a). https://github.com/SliSE-SC/SliSE. Accessed: April 2026.
[8] [n. d.]. ERC20 Token Standard. https://github.com/ethereum/ercs/blob/master/ERCS/erc-20.md. Accessed: April 2026.
[9] [n. d.]. Etherscan. https://etherscan.io. Accessed: April 2026.
[10] [n. d.]. LENT-SSE: Leveraging Executed and Near Transactions for Speculative Symbolic Execution of Smart Contracts (322bdf6317fa4fdc52e0cdd5303d309fb918965a). https://github.com/tczpl/lent-sse. Accessed: April 2026.
[11] [n. d.]. Mythril: a security analysis tool for EVM bytecode. https://github.com/ConsenSys/mythril. Accessed: April 2026.
[12] [n. d.]. PrettySmart (70edb8f310584d2e558a489a7c841df17195817a). https://github.com/Z-Zhijie/PrettySmart. Accessed: April 2026.

[13] [n. d.]. RLF (258c710362d12bc42653431725669c4080d76b62). https://github.com/Demonhero0/rlf. Accessed: April 2026.
[14] [n. d.]. Sailfish (bbab10cde68cc12a2f50e85a40dae213f5453259). https://github.com/ucsb-seclab/sailfish. Accessed: April 2026.
[15] [n. d.]. Smartian (badd4ffe9dd47270b6abbe27af84a6435263716c). https://github.com/SoftSec-KAIST/Smartian. Accessed: April 2026.
[16] [n. d.]. Solidity Documentation. https://docs.soliditylang.org. Accessed: April 2026.
[17] 2020. A GitHub repository for SmarTest. (36d191eca5e82e52297ed78b2cf8ff2ce509f7d8). https://github.com/kupl/VeriSmart-public. Accessed: April 2026.
[18] [n. d.]. An URL for the Z3's tactic `elim-uncnstr`. https://microsoft.github.io/z3guide/docs/strategies/summary/#tactic-elim-uncnstr. Accessed: April 2026.
[19] [n. d.]. Another URL for the Z3's tactic `elim-uncnstr`. https://z3prover.github.io/papers/z3internals.html. Accessed: April 2026.
[20] [n. d.]. Foundry: a blazing fast, portable and modular toolkit for Ethereum application development. https://github.com/foundry-rs/foundry. Accessed: April 2026.
[21] [n. d.]. The CVE report for the `Trabet_Coin` contract. https://www.cve.org/CVERecord?id=CVE-2018-13557. Accessed: April 2026.
[22] [n. d.]. The `Trabet_Coin` contract. https://etherscan.io/address/0x2f7f26470517c43cd142ecb9a75347454af74b00/. Accessed: April 2026.
[23] [n. d.]. Z3. https://github.com/Z3Prover/z3. Accessed: April 2026.
[24] Saswat Anand, Corina S Păsăreanu, and Willem Visser. 2009. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer* 11 (2009), 53–67.
[25] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. doi:10.1145/3182657
[26] Guillaume Ballet, Vitalik Buterin, and Dankrad Feist. 2023. EIP-6780: SELFDESTRUCT only in same transaction. https://eips.ethereum.org/EIPS/eip-6780. Accessed: April 2026.
[27] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 351–366.
[28] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*. 161–178. doi:10.1109/SP46214.2022.9833721
[29] Aaron R Bradley and Zohar Manna. 2007. *The calculus of computation: decision procedures with applications to verification.* Springer Science & Business Media.
[30] Yizhou Chen, Zeyu Sun, Zhihao Gong, and Dan Hao. 2024. Improving Smart Contract Security with Contrastive Learning-based Vulnerability Detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Article 156, 11 pages. doi:10.1145/3597503.3639173
[31] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 227–239. doi:10.1109/ASE51524.2021.9678888
[32] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.
[33] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 8–15. doi:10.1109/WETSEB.2019.00008
[34] Yu Feng, Emina Torlak, and Rastislav Bodik. 2021. Summary-based symbolic evaluation for smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1141–1152. doi:10.1145/3324884.3416646
[35] Klint Finley. [n. d.]. 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: April 2026.
[36] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. eTainter: detecting gas-related vulnerabilities in smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. 728–739. doi:10.1145/3533767.3534378
[37] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 945–956. doi:10.1109/ICSE48619.2023.00087

[38] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. 2015. Assertion guided symbolic execution of multithreaded programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. 854–865. doi:10.1145/2786805.2786841

[39] Sujin Han, Jinseo Kim, Sung-Ju Lee, and Insu Yun. 2025. Automated Attack Synthesis for Constant Product Market Makers. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA003 (June 2025), 22 pages. doi:10.1145/3728872

[40] Sicheng Hao, Yuhong Nan, Zibin Zheng, and Xiaohui Liu. 2023. SmartCoCo: Checking Comment-Code Inconsistency in Smart Contracts via Constraint Propagation and Binding. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 294–306. doi:10.1109/ASE56229.2023.00142

[41] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 531–548. doi:10.1145/3319535.3363230

[42] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. 2013. Boosting concolic testing via interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 48–58. doi:10.1145/2491411.2491425

[43] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. 2012. TRACER: A Symbolic Execution Tool for Verification. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 758–766.

[44] Joxan Jaffar, Jorge A. Navas, and Andrew E. Santosa. 2011. Unbounded symbolic execution for program verification. In *Proceedings of the Second International Conference on Runtime Verification* (San Francisco, CA) *(RV'11)*. Springer-Verlag, Berlin, Heidelberg, 396–411. doi:10.1007/978-3-642-29860-8_32

[45] Joxan Jaffar, Andrew E. Santosa, and Rundefinedzvan Voicu. 2009. An interpolation method for CLP traversal. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming* (Lisbon, Portugal) *(CP'09)*. Springer-Verlag, Berlin, Heidelberg, 454–469.

[46] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf

[47] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2019. Exploiting the laws of order in smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 363–373. doi:10.1145/3293882.3330560

[48] Ted Kremenek. [n. d.]. Attempting to speed up static analysis. https://lists.llvm.org/pipermail/cfe-dev/2015-August/044825.html. Accessed: April 2026.

[49] Wenkai Li, Xiaoqi Li, Zongwei Li, and Yuqing Zhang. 2024. COBRA: Interaction-Aware Bytecode-Level Vulnerability Detector for Smart Contracts. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. 1358–1369. doi:10.1145/3691620.3695601

[50] Yueqi Li, S. C. Cheung, Xiangyu Zhang, and Yepang Liu. 2014. Scaling Up Symbolic Analysis by Removing Z-Equivalent States. *ACM Trans. Softw. Eng. Methodol.* 23, 4, Article 34 (Sept. 2014), 32 pages. doi:10.1145/2652484

[51] Zeqin Liao, Sicheng Hao, Yuhong Nan, and Zibin Zheng. 2023. SmartState: Detecting State-Reverting Vulnerabilities in Smart Contracts via Fine-Grained State-Dependency Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. ACM, 980–991. doi:10.1145/3597926.3598111

[52] Zeqin Liao, Yuhong Nan, Henglong Liang, Sicheng Hao, Juan Zhai, Jiajing Wu, and Zibin Zheng. 2024. SmartAxe: Detecting Cross-Chain Vulnerabilities in Bridge Smart Contracts via Fine-Grained Static Analysis. *Proc. ACM Softw. Eng.* 1, FSE, Article 12 (July 2024), 22 pages. doi:10.1145/3643738

[53] Ye Liu, Yi Li, Shang-Wei Lin, and Cyrille Artho. 2022. Finding permission bugs in smart contracts with role mining. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 716–727. doi:10.1145/3533767.3534372

[54] Yinxi Liu, Wei Meng, and Yinqian Zhang. 2025. Detecting Smart Contract State-Inconsistency Bugs via Flow Divergence and Multiplex Symbolic Execution. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE002 (June 2025), 22 pages. doi:10.1145/3715712

[55] Chenyang Ma, Wei Song, and Jeff Huang. 2023. TransRacer: Function Dependence-Guided Transaction Race Detection for Smart Contracts. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 947–959. doi:10.1145/3611643.3616281

[56] Kenneth L. McMillan. 2010. Lazy Annotation for Program Testing and Verification. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 104–118.

[57] Yifan Mo, Jiachi Chen, Yanlin Wang, and Zibin Zheng. 2023. Toward Automated Detecting Unanticipated Price Feed in Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1257–1268. doi:10.1145/3597926.3598133

[58] Santiago Palladino. [n. d.]. The Parity Wallet Hack Explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/. Accessed: April 2026.

[59] Nikhil Parasaram, Earl T. Barr, Sergey Mechtaev, and Marcel Böhme. 2024. Precise Data-Driven Approximation for Program Analysis via Fuzzing. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, 611–623. doi:10.1109/ASE56229.2023.00185

[60] Priya. [n. d.]. Over $900K Stolen as Cybercriminals Exploit Vulnerabilities in Smart Contracts to Target Crypto Wallets. https://cyberpress.org/smart-contract-vulnerabilities/. Accessed: April 2026.

[61] Dawei Qi, Hoang D. T. Nguyen, and Abhik Roychoudhury. 2013. Path exploration based on symbolic output. *ACM Trans. Softw. Eng. Methodol.* 22, 4, Article 32 (Oct. 2013), 41 pages. doi:10.1145/2522920.2522925

[62] Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan Karame, and Lucas Davi. 2023. EFCF: High Performance Smart Contract Fuzzing for Exploit Generation. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 449–471. doi:10.1109/EuroSP57164.2023.00034

[63] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. EThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. 621–640. doi:10.1145/3372297.3417250

[64] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 322–333. doi:10.1145/3597926.3598059

[65] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/so

[66] S. So, M. Lee, J. Park, H. Lee, and H. Oh. 2020. VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. 718–734. doi:10.1109/SP.2020.00032

[67] Sunbeom So and Hakjoo Oh. 2023. SmartFix: Fixing Vulnerable Smart Contracts by Accelerating Generate-and-Verify Repair using Statistical Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 185–197. doi:10.1145/3611643.3616341

[68] Hyegeun Song, Jiseong Han, and Sunbeom So. 2026. SmarTrim: Symbolic Execution for Smart Contracts Powered by Redundant Transaction-Sequence Pruning. doi:10.5281/zenodo.19583475

[69] Hyegeun Song, Jiseong Han, and Sunbeom So. 2026. SmarTrim: Symbolic Execution for Smart Contracts Powered by Redundant Transaction-Sequence Pruning. https://github.com/ku-formal/SmarTrim-Artifact/tree/fse2026

[70] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. 2021. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*. 555–571. doi:10.1109/SP40001.2021.00085

[71] Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2023. Effectively Generating Vulnerable Transaction Sequences in Smart Contracts with Reinforcement Learning-guided Fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 36, 12 pages. doi:10.1145/3551349.3560429

[72] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 166, 13 pages. doi:10.1145/3597503.3639117

[73] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. 103–119. doi:10.1109/EuroSP51992.2021.00018

[74] Haijun Wang, Ting Liu, Xiaohong Guan, Chao Shen, Qinghua Zheng, and Zijiang Yang. 2017. Dependence Guided Symbolic Execution. *IEEE Transactions on Software Engineering* 43, 3 (2017), 252–271. doi:10.1109/TSE.2016.2584063

[75] Sally Junsong Wang, Kexin Pei, and Junfeng Yang. 2024. SmartInv: Multimodal Learning for Smart Contract Invariant Inference. In *2024 IEEE Symposium on Security and Privacy (SP)*. 2217–2235. doi:10.1109/SP54263.2024.00126

[76] Zexu Wang, Jiachi Chen, Yanlin Wang, Yu Zhang, Weizhe Zhang, and Zibin Zheng. 2024. Efficiently Detecting Reentrancy Vulnerabilities in Complex Smart Contracts. *Proc. ACM Softw. Eng.* 1, FSE, Article 8 (July 2024), 21 pages. doi:10.1145/3643734

[77] Yin Wu, Xiaofei Xie, Chenyang Peng, Dijun Liu, Hao Wu, Ming Fan, Ting Liu, and Haijun Wang. 2024. AdvSCanner: Generating Adversarial Smart Contracts to Exploit Reentrancy Vulnerabilities Using LLM and Static Analysis. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1019–1031. doi:10.1145/3691620.3695482

[78] Mingxi Ye, Xingwei Lin, Yuhong Nan, Jiajing Wu, and Zibin Zheng. 2024. Midas: Mining Profitable Exploits in On-Chain Smart Contracts via Feedback-Driven Fuzzing and Differential Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 794–805. doi:10.1145/3650212.3680321

[79] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2015. Postconditioned Symbolic Execution. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. doi:10.1109/ICST.2015.7102601

[80] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2018. Eliminating Path Redundancy via Postconditioned Symbolic Execution. *IEEE Transactions on Software Engineering* 44, 1 (2018), 25–43. doi:10.1109/TSE.2017.2659751

[81] Shuai Zhang, Meng Wang, Yi Liu, Yuhan Zhang, and Bin Yu. 2022. Multi-Transaction Sequence Vulnerability Detection for Smart Contracts based on Inter-Path Data Dependency. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. 616–627. doi:10.1109/QRS57517.2022.00068

[82] William Zhang, Sebastian Banescu, Leonardo Pasos, Steven Stewart, and Vijay Ganesh. 2019. MPro: Combining Static and Symbolic Analysis for Scalable Testing of Smart Contract . In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Los Alamitos, CA, USA, 456–462. doi:10.1109/ISSRE.2019.00052

[83] Wuqi Zhang, Zhuo Zhang, Qingkai Shi, Lu Liu, Lili Wei, Yepang Liu, Xiangyu Zhang, and Shing-Chi Cheung. 2024. Nyx: Detecting Exploitable Front-Running Vulnerabilities in Smart Contracts. In *2024 IEEE Symposium on Security and Privacy (SP)*. 2198–2216. doi:10.1109/SP54263.2024.00146

[84] Peilin Zheng, Bowei Su, Xiapu Luo, Ting Chen, Neng Zhang, and Zibin Zheng. 2024. LENT-SSE: Leveraging Executed and Near Transactions for Speculative Symbolic Execution of Smart Contracts. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 566–577. doi:10.1145/3650212.3680303

[85] Peilin Zheng, Zibin Zheng, and Xiapu Luo. 2022. Park: accelerating smart contract vulnerability detection via parallel-fork symbolic execution. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. 740–751. doi:10.1145/3533767.3534395

[86] Zhijie Zhong, Zibin Zheng, Hong-Ning Dai, Qing Xue, Junjia Chen, and Yuhong Nan. 2024. PrettySmart: Detecting Permission Re-delegation Vulnerability for Token Behaviors in Smart Contracts. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 168, 12 pages. doi:10.1145/3597503.3639140

Received 2025-09-12; accepted 2025-12-22