LENT-SSE: Leveraging Executed and Near Transactions for Speculative Symbolic Execution of Smart Contracts

Peilin Zheng
Sun Yat-sen University
China
zhengplin@mail.sysu.edu.cn

Bowei Su
Sun Yat-sen University
China
subw3@mail2.sysu.edu.cn

Xiapu Luo
The Hong Kong Polytechnic University
China
csxluo@comp.polyu.edu.hk

Ting Chen
University of Electronic Science and Technology of China
China
brokendragon@uestc.edu.cn

Neng Zhang
Sun Yat-sen University
China
zhangn279@mail.sysu.edu.cn

Zibin Zheng*
Sun Yat-sen University
China
zhzibin@mail.sysu.edu.cn

Abstract

Symbolic execution has proven effective for code analytics in smart contracts. However, for smart contracts, existing symbolic tools use multiple-transaction symbolic execution, which differs from traditional symbolic tools and also exacerbates the path explosion problem. In this paper, we first quantitatively analyze the bottleneck of symbolic execution in multiple transactions (TXs), finding the redundancy of the paths of TXs. Based on this finding, we propose LENT-SSE as a new speculation heuristic for Speculative Symbolic Execution of smart contracts, which leverages the executed and near TXs for skipping and recalling the SMT solving of paths. LENT-SSE uses an executed-transaction-based skipping algorithm to reduce the time required for SMT solving by leveraging the redundancy between executed and executing paths. Moreover, LENT-SSE uses a near-transaction-based recalling algorithm to reduce false skipping of the solving paths. Experimental results on the SmartBugs dataset show that LENT-SSE can reduce the total time by 37.4% and the solving time of paths by 65.2% on average without reducing the reported bugs. On the other dataset of 1000 realistic contracts, the total time and solving time are reduced by 38.1% and 54.7%.

CCS Concepts

• **Software and its engineering → Development frameworks and environments**; **Software reliability**; **Software safety**.

Keywords

Blockchain, Smart Contract, Symbolic Execution

*Corresponding Author.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680303

ACM Reference Format:
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 (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680303

1 Introduction

With the development of blockchain-based smart contracts, security issues and tools have attracted lots of attention. There are various techniques applied to code analytics of smart contracts, such as symbolic execution [12], fuzzing [19], and static analysis [7].

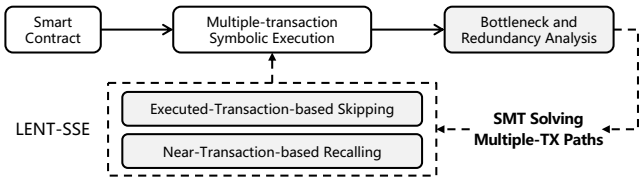


Figure 1: Overview of This Paper

The main idea of symbolic execution (SE) is to use symbols to traverse the execution path of codes [12]. The number of potential paths can be huge. That is called the path explosion problem. As for the symbolic execution of smart contracts, many tools have been proposed, e.g., Oyente [16], Mythril [18], Manticore [17]. Differing from traditional code, smart contracts are driven by multiple blockchain transactions (TXs), and different transactions share the same state. Therefore, when performing symbolic execution on smart contracts, it is needed to symbolize multiple transactions. This feature exacerbates the path explosion problem, as more TXs lead to more paths. In other words, path explosion is not only related to the complexity of the contract code itself, but also related to the number of symbolic transactions. The space of path explosion is exponential with the number of transactions.

Speculative Symbolic Execution (SSE) [34, 35] has been proposed to ignore checking branch feasibility with heuristics during SE, in order to accelerate the path exploration. However, previous SSE

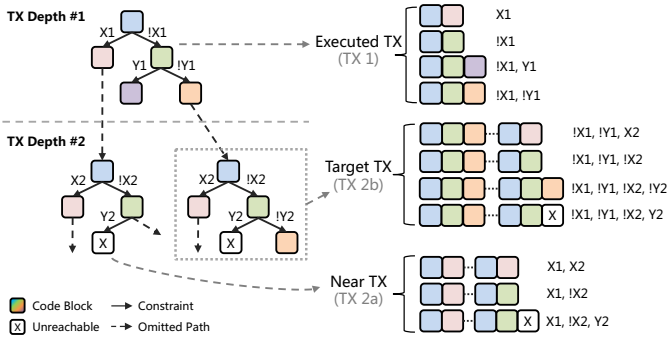


Figure 2: Example Paths and Constraints of Symbolic Execution in Multiple TXs of a Single Smart Contract

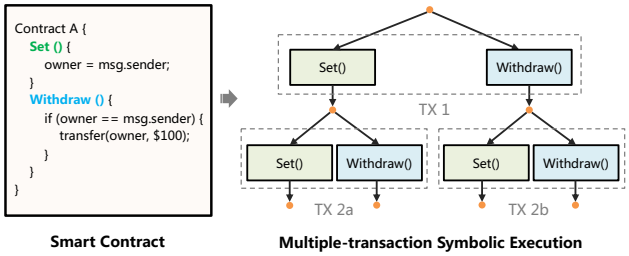


Figure 3: Background of Multiple TXs in SE of a Single Smart Contracts

methods/heuristics focus on traditional codes, hence do not consider the feature of multiple-TXs in smart contracts.

Addressing this issue, we propose LENT-SSE with the idea of Leveraging Executed and Near Transactions, which is a new heuristic for Speculative Symbolic Execution of smart contracts.

Figure 1 shows the overview of this paper. We first quantitatively analyze the bottleneck and redundancy of multiple-TX SE in smart contracts, and then propose LENT-SSE with two algorithms:

Bottleneck and Redundancy Analysis of SE in Multiple TXs. We use Mythril [18] (one of the popular symbolic tools of smart contracts) with the SmartBugs [6] dataset to measure the bottleneck. We find that 79.7% of the time in multiple-TX SE of smart contracts is solving constraints in SMT solver. Moreover, 73.8% of the SMT solving time is used to check path reachability, which is the bottleneck. We also measure the path redundancy between executed and executing transactions in each single contract. We find that, as for more than half of the contracts, 80% of the solving paths (not constraints) of a single contract are the same. As shown in Figure 2, the same code blocks are marked in the same colour, where three TXs have the same paths ("blue-pink", "blue-green", etc.) but with different constraints (in the right) to be solved.

Executed-Transaction-based Skipping. Based on the above findings, We propose LENT-SSE. It includes an Executed-Transaction-based Skipping Algorithm (ETSA). The main idea of ETSA is to speculatively execute the same paths of the executed transactions by optimistically skipping the SMT solving of them. As shown in Figure 2, in the target TX 2b, the first three paths are considered unnecessary to be solved as they have shown up in the executed TX, although their constraints are totally different in TX 1 and 2b. However, this also causes a new problem for false skipping, which falsely skips the solving and imports unreachable paths. LENT-SSE solves it with the next algorithm.

Near-Transaction-based Recalling. To prevent false skipping, LENT-SSE includes a Near-Transaction-based Recalling Algorithm (NTRA). For a target TX, in this paper, its near TXs refer to the TXs in the same TX depth. NTRA determines whether a path should not be skipped by searching for the potential unreachable paths in the near TXs. As shown in Figure 2, TX 2b learns from 2a. Hence, the unreachable path ("blue-green-×") in TX 2b is recalled to be solved to check the reachability. In this way, LENT-SSE can recall SMT solving of some paths to reduce wasting time on unreachable paths.

In short, the main novelty of LENT-SSE is reusing the SMT results of paths (NOT constraints) in different TXs of a single smart contract. A comparison between LENT-SSE and relevant tools (SSE [35], Green [28], GreenTrie [11]) will be given in §4.3.

We implement LENT-SSE on Mythril [18] and conduct reproducible experiments on two third-party datasets (SmartBugs and Top1k). The key results are as follows. **(1)** As for the effectiveness of LENT-SSE, on SmartBugs dataset [6], without reducing the reported bugs, LENT-Mythril can reduce the total time by 37.4% and the solving time of paths by 65.2%, compared with the original Mythril. On the dataset of realistic top 1000 contracts, the total time and solving time are reduced by 38.1% and 54.7%. **(2)** Compared to the existing Speculative Symbolic Execution (SSE) method [35], while SSE works well in a single transaction, LENT-SSE works better in multiple transactions. **(3)** As for the detailed improvement of each smart contract, we find that LENT-SSE accelerates more than 92.8% of the tested contracts. For specific complicated contracts, LENT-SSE can save the total time by more than 60%. **(4)** As for the impact of parameters, we find the timeout/rlimit (parameters of SMT solving), number of transactions, and complexity of contracts are the key parameters that affect the effectiveness. The details are given in §7. Codes and results are released on https://github.com/tczpl/lent-sse/.

Contribution. The contribution of this paper can be summarized as follows.

- We conduct quantitative analysis on the multiple-TX symbolic execution of smart contracts, finding the bottleneck and redundancy of multiple-TX paths.
- LENT-SSE proposes an Executed-Transaction-based Skipping Algorithm, levering the path redundancy of different transactions to reduce the times of SMT solving.
- LENT-SSE designs a Near-Transaction-based Recalling Algorithm to recall the SMT solving of the potential unreachable paths.

2 Background

Symbolic Execution (SE). Symbolic execution is a technique that uses symbols to cover several possible inputs of the program and comes out with the execution paths of the codes [1, 12]. There are several symbolic tools for traditional programs, e.g., KLEE [4], etc.

SMT Solving. Satisfiability Modulo Theories (SMT) is the problem of determining whether a mathematical formula is satisfiable [5]. SMT solving in SE is usually used to check path reachability. It transforms the path of a symbolic tree into a set of constraints and then tries to solve them. The path is reachable once the constraints

are satisfiable. Moreover, as later introduced, for detecting smart contract vulnerability, SMT solving not only happens in the path reachability but also happens in the vulnerability detection and exact transaction generation of smart contracts.

Speculative Symbolic Execution (SSE). Since checking path reachability costs much time, SSE [35] is proposed to save the time. The main idea is to ignore the reachability checking of the SMT solver until a limit is reached, with further improvement such as unsatisfiable core based backtracking and absurdity based optimization [34]. Previous SSE methods do not focus on smart contracts, while this paper proposes a new heuristic for SSE in smart contracts.

Blockchain-based Smart Contract. Blockchain-based smart contracts usually use the state-transition model. Each smart contract [3] on the blockchain has a state, which is operated by the transactions (TX) that trigger the contract. Therefore, the input of a contract is a sequence of transactions rather than a single transaction, which is different from the inputs to traditional programs.

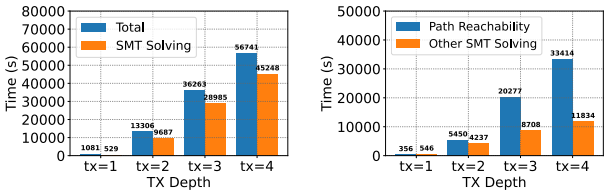
Multiple-TX Vulnerability. Some vulnerabilities in smart contracts require invoking a specific sequence of multiple transactions to trigger. Figure 3 shows an example of a vulnerable smart contract. Its funding can be stolen by the attacker's two specific transactions ("Set()" changes the owner to the attacker, then "Withdraw()" sends $100 from the contract to the attacker).

Multiple-TX SE of Contract. Since multiple-TX vulnerabilities exist, symbolic execution tools (including Mythril, Manticore, etc.) usually need to symbolize multiple transactions. As shown in Figure 3, starting with a root state, *TX 1* generates two (or more) states, and then *TX 2a* and *TX 2b* continue to execute on the previous state. During the SE, the SE tree (the right part in the figure) will be evaluated by the detecting modules to check whether there are vulnerabilities in the execution. Therefore, once the TX depth increases, the number of paths and states could be very large, which limits the effectiveness of SE of smart contracts.

Executed and Near TX. In this paper, as for a smart contract, when it is being executed in a TX that is in the depth N, the TXs executed before TX N (1, 2, ..., N-1) are called the Executed TXs, while the other TXs in the same depth (N) are called the Near TXs. For example, as shown in Figure 2 and Figure 3, *TX 1* is the executed TX of *TX 2a* and *TX 2b*. And *TX 2a* is the near TX of *TX 2b*, *TX 2b* is also the near TX of *TX 2a*.

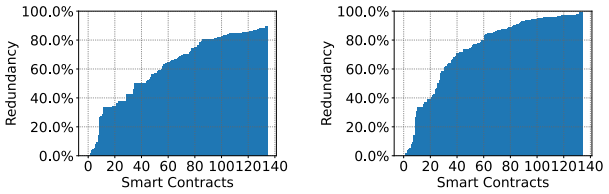
3 Bottleneck and Redundancy of Multiple-TX SE

In order to measure the bottleneck of SE in multiple TXs, we conduct a quantitative evaluation of the processes during SE. The following measurement is done by using Mythril [18] to validate the smart contracts in the SmartBugs-Curated dataset [6]. We instrument Mythril to record the detailed time costs and paths during the multiple-TX SE. SmartBugs-Curated Dataset consists of 142 smart contracts. 7 of them are out-of-memory or out-of-time when TX Depth is set as 4. Hence 142-7=135 of them are used for the following comparison of time costs and paths. Detailed setup and parameters (timeout, rlimit, tx, etc.) will be introduced in §7. The measurement is to answer two questions: **(1)** What costs the most time during multiple-TX SE? **(2)** How similar are the paths in different transactions?



(a) Total Time (b) SMT Solving Time

Figure 4: Distribution of Time in Multiple-TX SE



(a) TX Depth 2 (b) TX Depth 4

Figure 5: Redundancy of Path Solving in Multiple-TX SE

3.1 Distribution of Time

Distribution of Total Time. We measure the time of SMT solving to compare it with the total time. The results are shown in Figure 4(a). When TX depth is set as 1, the SE spends about half the time on SMT solving. However, when TX depth goes deeper, the ratio of SMT solving goes higher. Setting tx=4, the SMT solving takes up $\frac{45248}{56741} = 79.7\%$ of time in SE.

Distribution of SMT Solving. Moreover, we also measure the details of the SMT solving time. As mentioned in §2, the SMT solving not only happens in the path reachability. As shown in Figure 4(b), the SMT solving that is not checking the path reachability is named "Other SMT Solving". The "Other SMT Solving" includes the SMT solving time of vulnerability detection and exact transaction generation. Figure 4(b) shows that, when tx≥ 2, most SMT solving time is used on checking the path reachability. Setting tx=4, the path reachability takes up $\frac{33414}{45248} = 73.8\%$ of SMT solving time and $\frac{33414}{56741} = 58.9\%$ of total time .

Therefore, we can conclude that, in multiple-TX SE, the checking of path reachability costs the most time.

3.2 Path Redundancy of Different Transactions

Path. In this paper, a path in the symbolic execution tree is represented as a list of jump PC (program counter) and branches of the code blocks in the TX. Given the example in Figure 2, the path of blue-green-orange is regarded as one path in this paper. Note that a path is not referred to its constraints. The same paths (e.g., blue-green-orange) in different TXs are with different constraints. In the latter description, LENT-SSE does not cache or reuse the constraints but leverages the paths.

Redundancy of Solving Paths. We reviewed the history of the SMT solver to evaluate the redundancy of path-solving. Once the solver solves the constraints with a path that has shown up before in previous TXs (in the same single contract), then this path is regarded as a repeated path. The ratio of the number of repeated paths to the total number of paths is the redundancy rate of path solving. The results are shown in Figure 5. As shown in Figure 5(a)

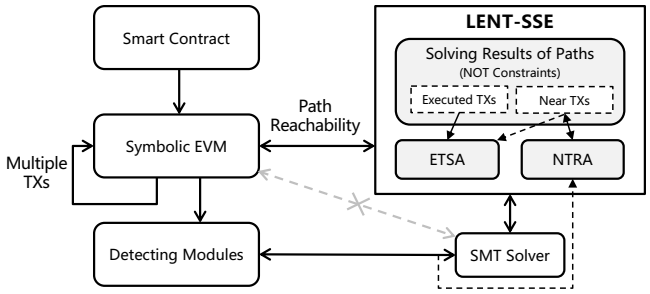


Figure 6: Architecture of LENT-SSE

and Figure 5(b), the redundancy of solving occurs in most contracts. Setting TX=4, for more than half of the contracts, 80% of the solving paths have appeared previously in their execution.

In summary, the bottleneck of SE in multiple TXs of smart contracts is the checking of path reachability. On the other hand, the redundancy of solving paths is significantly common in multiple TXs. Hence different TXs are with several redundant paths. Therefore, we leverage these two findings to design LENT-SSE.

4 Architecture of LENT-SSE

4.1 Overview

Goal. Based on the observation in §3, we design LENT-SSE to shorten the time of checking path reachability by leveraging the path redundancy of different TXs of the contract, which can be considered as a new heuristic for speculative symbolic execution in smart contracts. Meanwhile, the results of reported vulnerabilities should not be reduced. Aiming at this goal, we design LENT-SSE for skipping SMT solving of paths by executed and near TXs in multiple-TX symbolic execution of smart contracts. Figure 6 shows the architecture of LENT-SSE.

Roles. The roles in the Figure 6 are explained as follows. **(1) Smart Contract:** The given smart contract that needs to be analyzed. **(2) Symbolic EVM:** Ethereum Virtual Machine (EVM) is one of the most popular execution environments of smart contracts. Symbolic EVM is the symbolic engine of EVM contracts. **(3) Detecting Modules:** The modules that are used to check whether the current state/path is with the pattern of vulnerabilities. **(4) SMT Solver:** The solver that checks the reachability of paths and generates the exact assignment (model) of the symbolic input, etc. **(5) LENT-SSE:** The main contribution of this paper. It includes a space of memory to restore the paths and solving results. In LENT-SSE, each path is represented as a serial of executed EVM instructions. The solving results consist of the results of executed and near TXs. For ease of understanding, given the example in Figure 3 and Figure 2, as for the target *TX 2b*, *TX 1* could be its executed TX, and *TX 2a* could be its near TX. The Executed-Transaction-based Skipping Algorithm (ETSA) and Near-Transaction-based Recalling Algorithm (NTRA) leverage the information of the solving results. The details of the ETSA and NTRA will be introduced in §5 and §6, respectively.

4.2 Workflow

There are two workflows in Figure 6, as follows.

Original Workflow. In the workflow of original symbolic tools of smart contracts (e.g., Mythril, Manticore), contracts are input into the symbolic EVM. If there are multiple symbolic transactions, the output states will be input into the EVM as a loop. During the SE, the symbolic EVM will call the SMT solvers to check the path reachability (the gray dotted arrow with "×"). Moreover, the states are evaluated by the detecting modules to check vulnerabilities or get an exact assignment of the transaction input (the bottom solid arrow).

LENT-SSE Workflow. The difference between LENT-SSE and the original workflow is the checking of path reachability. As shown in the figure, the gray dotted arrow is replaced with LENT-SSE. Once the symbolic EVM requires path reachability, it will ask LENT-SSE rather than the SMT solver. LENT-SSE will decide whether it should use the SMT solver to solve the constraints or not. By leveraging the historical solving results of paths in the ETSA and NTRA, the times of solving in redundant paths will be reduced. Note that even LENT-SSE "falsely" marks the unreachable path as reachable. If there is a vulnerability, the detecting modules will ask the SMT solver again to get the assignment. Hence it is like a "double check" that the "false skipping" would not import more false positives in the final reported bugs. Moreover, once the SMT solver finds some unreachable paths given by the detecting modules, it will also report to LENT's NTRA as solving results. A concrete example that shows how ETSA and NTRA work will be given in §6.

Benefits. The benefits of the LENT-SSE workflow are as follows. **(1) Saving time of redundant path solving:** Some redundant path solving can be blocked by the ETSA in LENT-SSE. Hence the time for redundant solving is saved. **(2) Reducing false skipping:** Some false skipped paths are recalled by the NTRA, reducing the effects of importing unreachable paths. **(3) Consistency with the original tool:** LENT-SSE does not cut the paths. Hence consistency with the original tools of vulnerability detection can be ensured.

Extra Costs. *(1) Memory:* Note that LENT-SSE does not cache the SMT constraints or opcodes as paths. It only takes the Program Counter of JUMP and the "LEFT/RIGHT" flag as paths. Hence each path is a short array of simple integers, which costs little memory. *(2) Time:* LENT-SSE requires more time in checking the historical paths. And it also requires more time in double checking the "false skipping" paths, as the approach may increase the time of vulnerability detection and exact transaction generation. The quantitative evaluation will be shown in §7.

Completeness and Soundness. (1) As for the completeness, since LENT-SSE does not prune the paths but skips the path-solving, it will explore more paths than the original once "false pruning" happens. Therefore, the LENT-SSE paths actually include the original paths. Hence completeness can be ensured. (2) As for the soundness, once "false skipping" happens, as shown in Figure 6, the detecting modules will double check the path. Therefore, the "false skipping" paths will not increase the final reported bugs/vulnerabilities of the contracts. §7.1 will show the results of the covered states and detected bugs to evaluate the completeness and soundness.

4.3 Comparison

In previous SE studies, the optimization of SMT solving has been studied. The relevant works are compared as follows.

LENT-SSE vs SSE. As mentioned in §2, the original SSE methods [34, 35] focus on traditional programs rather than smart contracts, hence does not leverage the multiple-TX information. As for

quantitative evaluation, §7.2 implements SSE on smart contracts and provides a quantitative comparison between LENT-SSE and SSE.

LENT-SSE vs Green/GreenTrie. Green [28] reuses the results of the solving constraints. And GreenTrie [11] reuses the results of the main components of constraints. Their main idea is that the constraints from various paths, once transformed, may become equivalent or inclusive, enabling the reuse of previously solving results. Unfortunately, they can hardly be adapted onto smart contracts. Because the contract constraints are usually different and can hardly transformed: **(1) Different constraints.** In the paths of contracts, the same paths are with different constraints. As shown in Figure 2, given "blue-pink", the constraints are "X1", "X1, X2", "!X1, !Y1, X2", respectively, as the TXs are started from different symbolic states. **(2) Difficult transformation.** Green/GreenTrie transforms the constraints (slicing and canonization) to the same form for reuse. However, in smart contracts, constraints involve some advanced features of SMT solver (e.g., Z3.Store, Z3.Function [5]), but Green/GreenTrie have not supported, making it hard to be sliced or canonized without further efforts. Hence, we cannot implement Green/GreenTrie for quantitative comparison. In short, the significant difference between LENT-SSE and Green/GreenTrie is that LENT-SSE uses the paths rather than the constraints. Moreover, Green/GreenTrie have also shown the reuse of constraints across programs, but LENT-SSE only reuses the paths in a single contract.

LENT-SSE vs Others. There are also other studies proposed to prune the paths, e.g., pruning paths in event-driven GUI programs [10], pruning paths in smart contracts [32], etc. Nevertheless, to the best of our knowledge, LENT-SSE is the first study that leverages the path redundancy of multiple TXs to skip the SMT solving rather than prune the paths. The detailed related works will be given in §8.

5 Executed-Transaction-based Skipping

This section details the example, idea, and algorithm of the Executed-Transaction-based Skipping Algorithm (ETSA).

Motivating Example. The motivating example of ETSA is shown in Figure 7. These two transactions are extracted from Figure 2. TX 1 is the first symbolic transaction, and TX 2b is executed based on the output state of TX 1. They both have the paths of "blue-pink", "blue-green", and "blue-green-orange". The reason is that although these two transactions start from different states, the differences between the states could be few. Hence their execution tree could share several same paths. This also corresponds to the proposed finding in §3. Therefore, after executing TX 1, when executing TX 2b, the green arrow in Figure 7 can be optimistically regarded as reachable to save time for SMT solving. Note that the dotted areas are with same paths but different constraints.

Main Idea. The main idea of ETSA is to leverage the similarity of the paths in different TXs. ETSA models each path as a list of PC (Program Counter) and branch (left or right) and then records the solved reachable paths. In further execution, once the same path shows up, the ETSA regards it as reachable without solving its constraints.

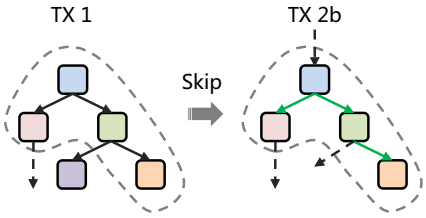


Figure 7: Motivating Example of ETSA

Algorithm 1: Pseudocode of ETSA

```
   Input: Executed TX Paths (ETXP)
   Input: Recalled Near TX Paths (RNTXP) [Optional]
1  /* Symbolic Execution in TX i */
2  for state ⟵pop states do
3      /* Symbolic execution of a state*/
4      newStates = INSTRUCTION(state)
5      for newState ∈ newStates do
6          newState.path = state.path
7          /* Path reachability*/
8          if newStates.length > 1 then
9              newState.path.push(<state.PC, newState.index>)
10             if newState.path ∈ ETXP-RNTXP then
11                 states ⟵add newState
12             else
13                 res = SOLVE(newState.constraints)
14                 if res = SAT then
15                     ETXP ⟵add newState.path
16                     states ⟵add newState
17                 end
18             end
19         else
20             states ⟵add newState
21         end
22     end
23 end
```

Algorithm. The pseudocode of ETSA is shown in Algorithm 1. Its input is a global set of Executed Transaction Paths (ETXP) by previous transactions. This set consists of the reachable paths that are solved in previously executed transactions. The ETXP is initialized as an empty set in TX 0. As shown in line 1 to line 4 of the algorithm, the symbolic execution engine withdraws a state from the state queue (states) to execute it. After that, new symbolic states are generated. The paths of the new states are inherited and copied from the previous state, as shown in line 6. If and only if the number of new states is greater than 1, the symbolic execution tree is forked into two branches. Then, the path reachability of the two branches needs to be checked. As shown in line 8, the path is constructed as a list of several pairs. Each pair consists of the PC of the previous state and the index of the new state, as shown in the following equations.

$$Path(S') = Path(S) + < S.PC, S'.index > \quad (1)$$

where S and S' represent the old and new states, and S'.index represents whether the path goes to "left" (index=0) or "right" (index=1). Finally, ETSA obtains the paths as follows.

$$Path(S) = [< PC_1, LEFT >, < PC_2, RIGHT >, ...] \quad (2)$$

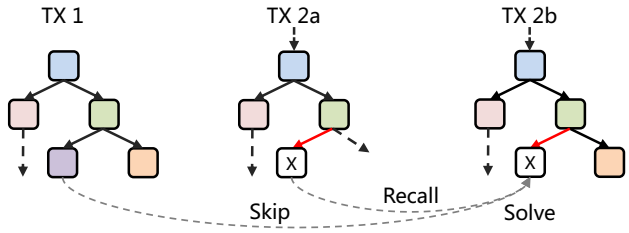


Figure 8: Motivating Example of NTRA

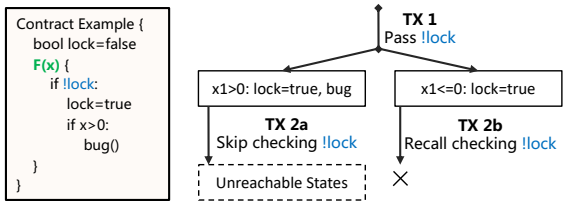


Figure 9: Concrete Contract Example of NTRA

If the path is in the ETXP but not in the RNTXP, then it is regarded as reachable, as shown in line 10 to line 11. Otherwise, the constraints of the new state will be checked by the SMT solver's "SOLVE()". If the constraints are satisfiable (SAT), then this path is reachable and will be added to the ETXP, as shown in line 12 to line 18.

Note that ETSA imports an optional set of RNTXP, which is a set maintained in the NTRA (introduced in the next section). RNTXP can be considered a global set that gets updated in the NTRA. If the NTRA is used, the "SOLVE()" leads to invoking NTRA as a wrapper of the solver. If NTRA is not used, the RNTXP is an empty set, then ETXP-RNTXP=ETXP in line 10.

Limitation. The main idea of ETSA is simple. However, although it shows effectiveness in reducing the time costs of multiple-TX SE (evaluated in §7), it cannot work well on some complicated contracts because ETSA will introduce more unreachable paths into the SE tree. Hence we need to recall some paths to solve their reachability, importing the RNTXP. The generation of RNTXP will be introduced in the next section.

6 Near-Transaction-based Recalling

This section details the Near-Transaction-based Recalling Algorithm (NTRA).

Motivating Example. The motivating example of NTRA is shown in Figure 8. These three transactions are also from Figure 2. The TX 1 is the first symbolic transaction, and the TX 2a and 2b are executed based on the output states of TX 1. As for TX 1 and TX 2b, if we use the ETSA to skip the solving of paths, we will regard the path "blue-green-purple" as reachable in TX 2b. However, its exact reachability can be false, as the real path should be "blue-green-×", as shown in the TX 2a and TX 2b. In other words, as shown in Figure 2, when "!X1, Y1" is satisfiable, "X1, !X2, Y2" and "!X1, !Y1, !X2, Y2" are not. If we consider "blue-green-purple(×)" as reachable in TX 2b, then further execution is needed and costs time. Note that the SMT solving not only happens in the path reachability, but also happens in the vulnerability detection and exact transaction generation. Therefore, once there are potential vulnerabilities, TX 2a has the opportunity to check the reachability of "blue-green-purple", and then it will find the path in TX 2a unreachable. Hence

Algorithm 2: Pseudocode of NTRA

```
   Input: Queries of SMT solver (inputs)
   Output: Query Results (outputs); Recalled Near TX Paths (RNTXP)
1  /* Wrapper of SMT Solver in Detecting Modules */
2  for constraints ← inputs do
3      state = STATE_OF(constraints)
4      res = SOLVE(constraints)
5      if res == SAT then
6          if state.path ∈ RNTXP then
7              RNTXP.remove(state.path)
8          end
9      else
10         if res == UNKNOWN then
11             /* Recall the solved unknown paths */
12             subpath = state.path
13             while subpath.length > 0 do
14                 RNTXP.add(subpath)
15                 subpath.pop()
16             end
17         end
18     end
19     outputs ← res
20 end
```

TX 2b can leverage this information to check out the "blue-green-purple(×)". This is just a simple motivating example. In the real SE of the contract, the sub-tree brought by the unreachable paths could be large. We will evaluate the wasting of time in §7. In short, as shown in the figure, for TX 2b, the red arrow should not be skipped by the path in TX 1 but should be recalled by the path in TX 2a.

Concrete Example. Figure 9 shows a concrete contract example to explain NTRA. When executing it with TX-depth=2, two symbolic inputs are marked as x1 and x2. In TX 1, we get two symbolic states by symbolic x1. In TX 2 (2a/2b), we execute on the above two states. First we execute TX 2a on (x1>0 | lock=true, bug), when considering the statements below "if !lock:" with its all constraints, since (!lock) previously has been passed in TX 1, as designed in ETSA, the reachability of state of (x1>0: lock=true, & !lock) will not be checked at TX 2a. Therefore, it will get two bugs (triggering the bug() two times from TX 1 and TX 2a). However, when NTRA checks the second bug with(x1>0: lock=true, & !lock & x2>0), it will find it unreachable. Therefore, next time, we execute TX 2b on (x1<=0: lock=true), when considering the "if !lock:" again, NTRA will recall and check the (x1<=0: lock=true, & !lock) , find it unreachable and avoid wasting more time. This example "lock" is widely used in smart contracts to prevent the famous Reentrancy Attack [16]. In this example, if we delete the statement "lock=true", then ETSA can work well alone, and NTRA can be not needed.

Main Idea. The main idea of NTRA is also to leverage the redundancy of the paths in different TXs. However, in contrast to ETSA, the purpose of NTRA is to recall the paths to prevent importing unreachable paths. It maintains a set of recalled paths by the near transactions. If a path cannot be reached in the near transactions, it will be added to the set to prevent the next skipping of its SMT solving. Moreover, all the sub-paths of the path are also added suspiciously. In this way, the next transaction will check the reachability of the path to avoid further useless execution. As the motivating example shows, this main idea has no conflict with the ETSA, since ETSA only skips the SMT solving of the path reachability, but NTRA also leverages the solving results of the vulnerability detection and

Table 1: Comparison between LENT-SSE and original tool on SmartBugs Dataset (RQ1)

Config	#Con'	Type	Bugs	TP	States (Δ)		Time (Δ)		Solved'	Path' Time (Δ)		Vul' Time (Δ)	
tx=2 timeout=5e3	138	Origin	553	98	739937	-	49072.3	-	37007	23931.9	-	25140.4	-
		LENT-SSE	554	99	780768	5.5%	32750.8	-33.3%	12663	6598.8	-72.4%	26152.1	4.0%
tx=2 rlimit=10e6	138	Origin	513	95	670231	-	24626.5	-	33734	10451.6	-	14174.9	-
		LENT-SSE	522	95	691925	3.2%	17596.4	-28.5%	12610	2947.6	-71.8%	14648.8	3.3%
tx=3 timeout=3e3	135	Origin	501	97	1714669	-	201342.3	-	102127	120972.4	-	80369.8	-
		LENT-SSE	511	99	1825445	6.5%	131809.4	-34.5%	37656	45388.4	-62.5%	86421.0	7.5%
tx=3 rlimit=5e6	137	Origin	449	90	1139947	-	47483.2	-	68948	25550.0	-	21933.3	-
		LENT-SSE	460	93	1187130	4.1%	29727.4	**-37.4%**	26063	8901.4	**-65.2%**	20826.0	-5.0%
tx=4 timeout=5e3	136	Origin	556	99	3911818	-	872543.9	-	258782	615298.2	-	257245.7	-
		LENT-SSE	561	99	4228968	8.1%	581569.0	-33.3%	98256	286491.4	-53.4%	295077.6	14.7%
tx=4 rlimit=5e6	135	Origin	432	88	1555923	-	56740.6	-	99686	33413.8	-	23326.8	-
		LENT-SSE	441	91	1597912	2.7%	39394.8	-30.6%	33646	14726.1	-55.9%	24668.7	5.8%

exact transaction generation from the near transactions, as shown in the bottom dotted line in Figure 6.

Algorithm. The pseudocode of NTRA is shown in Algorithm 2. The NTRA will take over and forward all the queries of the SMT solver, including the solving of symbolic EVM and the solving of detection modules. Therefore, it can be implemented as a wrapper of the SMT solver. As shown in line 1 to line 4, it solves the constraints by the SMT solver and gets the result. If a recent (near) transaction finds a recalled path that is reachable, then the path will be removed from the recalled set (RNTXP). Since the processing of constraints by detection modules is incremental, if the added constraints can be satisfiable, then the constraints of the path before the increase can also be satisfiable. Hence the path can be removed from the RNTXP, as shown in line 5 to line 8. If the result is unsatisfied (UNSAT), then it does not indicate that the constraints of the paths of near TXs are unsatisfied, since the UNSAT could result from the extra constraints added by detection modules. Hence nothing is done with the UNSAT result. If the result is unknown (UNKNOWN is a returned value of the SMT solver), then we consider the path beyond the solving power of the SMT solver (e.g., timeout). Note that, in smart contract SE, once a path is UNKNOWN, then its exact assignment is not available. And then the path cannot be used in further vulnerability detection or exact transaction generation. Hence it is unreachable, and should be stopped earlier next time (next transactions). LENT-SSE regards all the sub-paths as suspicious for exceeding the solving power. The calculation of the sub-paths is as follows.

$$SubPaths(S) = \{Path(S)[0:i]\}\,(i > 0) \tag{3}$$

where SubPaths represents the set of sub-paths of state S. NTRA adds all the sub-paths into the RNTXP, as shown in line 10 to line 16.

In this way, the skipped solving of the paths can be recalled in the next near transactions, as NTRA updates the RNTXP as a global set. Sincerely, the recalled strategy of the sub-paths could be different and complex in future work.

7 Evaluation

Implementation. As the architecture shown in §4, we implement LENT-SSE on Mythril. As surveyed in [36], Mythril is one of the most popular symbolic execution tools of smart contracts in both academia and industry. It enables multiple-transaction symbolic execution through the state-transition SE model. Therefore, we choose Mythril to implement the modules and algorithms (ETSA and NTRA) of LENT-SSE. Moreover, as the comparison with the Speculative Symbolic Execution (SSE) [35], we also implement the main idea of SSE on Mythril. We mainly modify the modules of symbolic EVM (sym.py), SMT solving (model.py), etc. Codes and results are released on https://github.com/tczpl/lent-sse/.

Dataset. We use the two datasets. *(1) SmartBugs [6]:* It includes 142 vulnerable Solidity contracts in 10 categories. The labelled contracts (SB-Curated) could provide a fair comparison between Mythril and LENT-Mythril by counting the true positives ("TP"). *(2) Top1K:* We get the top 1,000 contracts that are most triggered by users via 410,121,272 transactions among 573,785,320 contract-triggering transactions on Ethereum, proposed by XBlock-ETH [37]. Hence these contracts are popular contracts in the real world.

Parameters. There are several parameters in running the SE of smart contracts in this paper. *(1) Timeout/Rlimit:* The "timeout" and "rlimit" represent the limit of the Z3 solver[5]. They are the limits of time or resources (time+space) provided by the Z3 solver, preventing the solver from infinite solving time. Note that only one of these two parameters needs to be given to limit the solver. *(2) TX Depth:* The "tx" represents how many transactions are executed during the SE, namely the transaction depth. For example, the transaction depth in Figure 2 is 2. *(3) Memory:* For each run, we restrict the Python run-time with 4G memory at maximum.

Experimental Setup. We run the Mythril and LENT-Mythril on the server that is equipped with the Intel 10700F CPU and 32GB memory. Each run takes up one CPU core during symbolic execution using the depth-first-search strategy. Therefore, there are eight running processes in the meanwhile on the server, both for the original and LENT-based methods. Once complicated contracts with multiple transactions are running, the memory is not enough. In this case, in order to provide a fair comparison for Mythril and LENT-Mythril, as for the following evaluation, those contracts will not be counted in the comparison.

Common Metrics. There are two common metrics in the evaluation. *(1) Total Time:* The total symbolic execution time for a smart contract. The total time of a group of contracts is the sum of the total time of them. *(2) Path Reachability Time:* The time costs for checking the constraints of all paths by SMT solvers for a smart contract. The path reachability time of a group of contracts is the sum of their path reachability time. In the following evaluation, the "Path Reachability Time" is abbreviated as "Path' Time".

Table 2: Comparison between LENT-SSE and original tool on Top1K (RQ1)

Config	Type	States	Time (s)	Path' Time (s)
tx=2 rlimit=2e6 (998 con')	Origin	6967705	199509.15	126524.18
	LENT-SSE	7687630	157922.24 (-20.84%)	79083.67 (-37.49%)
tx=3 rlimit=2e6 (996 con')	Origin	8497453	272336.05	175787.39
	LENT-SSE	9692417	206663.97 (-24.11%)	100646.58 (-42.75%)
tx=4 rlimit=2e6 (994 con')	Origin	9512761	317178.68	208701.41
	LENT-SSE	11015938	234617.13 (-26.03%)	114107.49 (-45.33%)
tx=2 rlimit=8e6 (989 con')	Origin	13907079	790813.59	558653.33
	LENT-SSE	14621376	489405.77 **(-38.11%)**	253034.31 **(-54.71%)**

Research Questions. We first evaluate the effectiveness of LENT-SSE and then investigate the detailed improvement, with the following research questions (RQs): **RQ1.** Is LENT-SSE effective in reducing the time of multiple-transaction symbolic execution? **RQ2.** How do LENT-SSE and SSE perform in smart contracts? **RQ3.** How is the detailed improvement of the multiple-transaction symbolic execution in smart contracts? **RQ4.** What are the key parameters of the performance?

7.1 RQ1: Effectiveness of LENT-SSE

Motivation. The major target of LENT-SSE is to shorten the time in the multiple-TX SE in smart contracts. Hence we need to evaluate the effectiveness of the proposed algorithms.

Approach. First, using SmartBug and Top1K datasets, we run LENT-Mythril (LENT-SSE) and the original Mythril (Origin) in the same configuration to evaluate the performance. Second, we also run two versions of LENT-SSE: one is with NTRA ("ETSA+NTRA"), and the other is not ("ETSA"), in order to check out the effects of NTRA. Besides the total time and path reachability time (Path' Time), we also count the number of contracts (# Con'), total state, the number of positives (Bugs, which are the reported vulnerabilities), and the solved paths (Solved'). The total SMT solving time is composed of three parts: path reachability, vulnerability detection, and exact transaction generation. We mark the time of vulnerability detection and exact transaction generation as "Vul' Time". And we can consider all the time without the Path' Time is the Vul' Time. Furthermore, we calculate the change (Δ) of the states and time (compared with the original Mythril) to see whether the algorithms reduce the time costs of the SE.

Results. The results of SmartBugs and Top1K dataset are shown in Table 1 and Table 2. Note that the number of contracts in each group can be different due to the reason mentioned in the *Experimental Setup*. As shown in the configuration of "tx=3, rlimit=5,000,000", the total time is reduced by 37.4%. This reduction mainly comes from the reduction of the path reachability time, as ETSA+NTRA reduces the path reachability time by 65.2%. Moreover, since LENT-SSE does not skip the paths but skips the SMT solving, the number of total states, reported positive bugs ("Bugs"), and true positives ("TP") are not reduced compared to the original tool. The number of states that NTRA cannot prune (unreachable paths imported by ETSA) is reflected in the increment of the states in LENT-SSE. The percentages vary from 2% to 8%. Hence, it does not cost much more time. Although it still costs some time, LENT-SSE can save more time in solving paths (Δ Path' Time). In a macro view, the LENT does introduce extra time on vulnerability detection and exact transaction generation (Δ Vul' Time). But the extra percentages are less than 15% (with some possible margin of error), which does not have much effect on the total improvement. As shown in Table 2, LENT-SSE also works on the Top1K dataset, demonstrating its feasibility in real-world smart contracts. In the configuration of "tx=2, rlimit=8,000,000", the total time is reduced by 38.1% while the solving time is reduced by 54.7%.

Table 3: Comparison between ETSA and ETSA+NTRA in LENT-SSE (RQ1)

Config	Type	States	TP	Time (s)	P' Time (s)
tx=2 timeout=1e3 (137* con')	Origin	491721	89	13525.48	5594.75
	ETSA	663093	91	14309.09 (+5.79%)	1509.02 (-73.02%)
	ETSA+ NTRA	529753	90	10984.36 (-18.78%)	2181.92 (-61.00%)
tx=2 rlimit=10e6 (138 con')	Origin	670231	95	24626.47	10451.58
	ETSA	844095	95	25606.97 (+3.98%)	3501.67 (-66.49%)
	ETSA+ NTRA	691925	95	17596.37 (-28.55%)	2947.57 (-71.80%)
tx=3 timeout=1e3 (135 con')	Origin	857743	89	41829.94	24897.42
	ETSA	2375561	91	102815.79 **(+145.79%)**	5740.93 (-76.94%)
	ETSA+ NTRA	917448	89	29738.74 **(-28.90%)**	11071.59 (-55.53%)

As shown in Table 3, when LENT-SSE only uses ETSA without NTRA, the total time cannot be shortened, but the path reachability time can be reduced. The reason is that the ETSA is effective in reducing the solved paths but also imports unreachable paths. Hence, in some specific contracts, the total time could be longer once there are too many unreachable paths imported, leading to an increase in total time (up to +145.79%). As shown in Table 3, more time and states in ETSA can certainly detect more bugs, but our goal is to reduce the time. A detailed evaluation on ETSA and ETSA+NTRA will be given in §7.3. (In the first group, one contract in ETSA runs out of memory, hence it is counted in Table 1 but not counted in Table 3.)

Answer to RQ1. On the SmartBugs dataset, without reducing the reported bugs and true positives (effective buggy paths), LENT-Mythril reduces the total time by 37.4% and the solving time of paths by 65.2%, compared with the original Mythril. On the Top1K dataset, the total time and solving time is reduced by 38.1% and 54.7%, respectively.

7.2 RQ2: LENT-SSE and SSE

Motivation. A comparison between LENT-SSE and SSE is needed to evaluate the performance. Moreover, a performance discussion of the optimization of SSE will be given in §10.

Approach. As mentioned in §4.3, since Speculative Symbolic Execution (SSE) [35] has not been applied to smart contract SE, we implement it on Mythril for evaluation. The parameter of the Speculation Depth Limit (SSE-limit) is set to 2, 4, 8, 16, 32, 64. The rlimit is set as 2,000,000. We count the contracts with the shortest

Table 4: Comparison between LENT-SSE and SSE on SmartBugs (RQ2)

TX'	#Con'	SSE		# Shortest Time		
		Limit	Time (s)	Origin	SSE	LENT-SSE
1	137	2	1491.6	30	107	-
		4	1593.3	24	113	-
		8	2011.9	30	107	-
		16	3909.9	34	103	-
		32	5203.3	34	103	-
		64	5522.1	33	104	-
2	131	2	12063.9	10	15	106
		4	12917.3	9	15	107
		8	14408.7	8	25	98
		16	16682.9	9	25	97
		32	18484.5	9	34	88
		64	20367.1	9	40	82
3	125	2	42289.3	9	9	107
		4	48636.7	10	10	105
		8	53471.1	10	14	101
		16	68607.0	10	19	96
		32	88407.0	10	23	92
		64	94784.4	10	24	91
4	122	2	63187.2	15	15	92
		4	75747.5	15	16	91
		8	86800.1	14	17	91
		16	110831.1	15	21	86
		32	143568.1	16	30	76
		64	150918.4	16	28	78

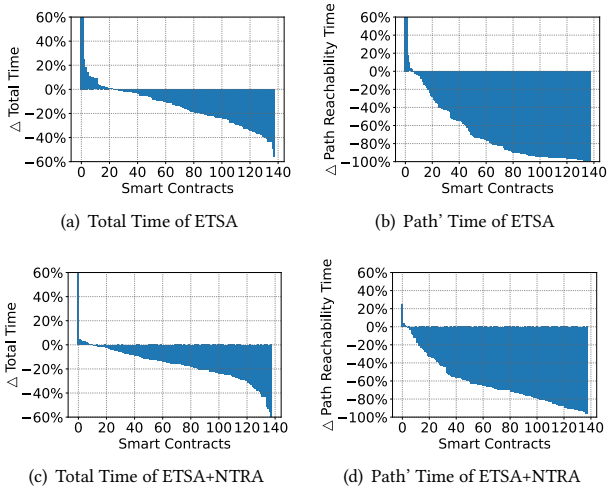
time (best performance) of each type (Origin, SSE, LENT-SSE) to compare the performance between LENT-SSE and SSE.

Results. The results of the SmartBugs dataset are shown in Table 4. Setting tx=1, LENT-SSE cannot work in the single symbolic TX, and SSE does work well in the single symbolic TX. As shown in Table 4, SSE reduces most contracts with different settings of the Speculation Depth Limit. The reason is that, in some contracts that are with few paths and few vulnerabilities (aka simple contracts), skipping more path reachability checking, the time will be shortened. Note that higher SSE-limits will make the complex contracts consume more states and time, also result in more out-of-memory contracts and reduce the intersection set for comparison. Therefore, we stop at SSE-limit=64. We can also find that, with an increase in SSE-limit, the overall time of SSE also increases, indicating worse performance. This results from extra exploration in the complex contracts (more paths and vulnerabilities than the simple contracts). In other words, with the increase of SSE-limit, SSE works better in simple contracts but worse in complex contracts. Given multiple TXs (tx=2, 3, 4), LENT-SSE performs better in most cases as it gets the shortest time in most contracts.

Answer to RQ2. On the SmartBugs dataset, while Speculative Symbolic Execution (SSE) works well in a single transaction (tx=1), LENT-SSE works better in multiple transactions (tx>1).

7.3 RQ3: Detailed Improvement of Contracts

Motivation. RQ1 and RQ2 only provide an overall improvement of LENT-SSE. However, the detailed improvement to specific contracts is unclear. There could be some contracts that affect the results significantly. Therefore, evaluation is needed to check the detailed improvement of each smart contract in the SmartBugs dataset.



(a) Total Time of ETSA (b) Path' Time of ETSA

(c) Total Time of ETSA+NTRA (d) Path' Time of ETSA+NTRA

Figure 10: Detailed Improvement of Contracts (RQ3)

Approach. Setting tx=2, rlimit=10,000,000, we run LENT-Mythril (ETSA, ETSA+NTRA) on the SmartBugs dataset to measure the detailed reduction of time compared with the original Mythril.

Results. Figure 10 shows the detailed improvement of the contracts. Figure 10(a) and Figure 10(b) show the detailed change of time in ETSA. Although the ETSA saved the total time and path reachability time for most contracts, there are still more than 20 contracts in which the total time is increased. This leads to the negative effects of acceleration corresponding to Table 3. As for ETSA+NTRA, only 10 contracts cannot be accelerated, as shown in Figure 10(c) and Figure 10(d). Hence, LENT-SSE can reduce the time costs of $\frac{138-10}{138} = 92.8\%$ of contracts. In the next subsection, we will explore the impact of the complexity of contracts. We find that the contracts with a reduction of more than 60% total time are executed for a longer time than others. Hence most of these contacts with better acceleration are regarded as complicated contracts.

Answer to RQ3. On the SmartBugs dataset, setting tx=2, LENT-SSE accelerates more than 92.8% of the tested contracts. For specific complicated contracts, LENT-SSE can save the total time by more than 60%.

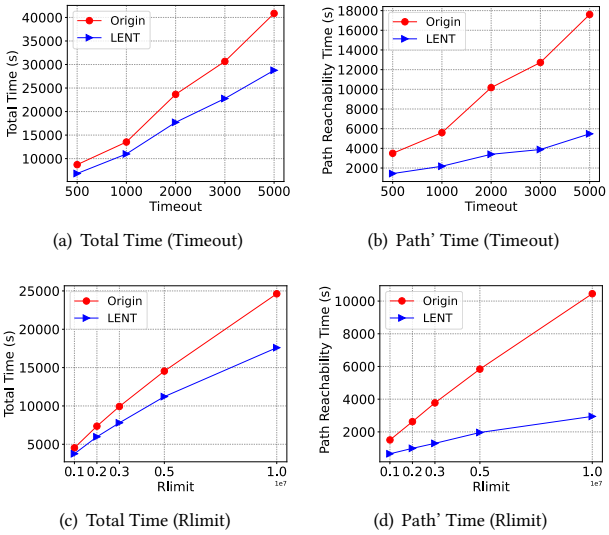
7.4 RQ4: Impact of Parameters

Motivation. The effectiveness and feasibility of SE tools vary from different configurations. Therefore, it is necessary to investigate the impact of parameters. In this subsection, we divide the RQ4 into three sub-RQs, investigating the impact of the timeout/rlimit, transaction depth, and contract complexity, respectively.

RQ4-1: What is the impact of the timeout/rlimit?

Approach. We respectively set tx=2 to run LENT-Mythril and Mythril in different "timeout" and "rlimit" of Z3 solver on the SmartBugs, then calculate the total time and path reachability time.

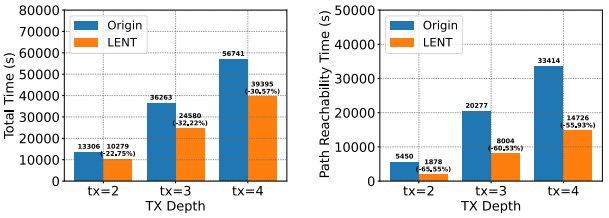
Results. The results are shown in Figure 11. Setting the timeout from 500 to 5000 (milliseconds), LENT-SSE can achieve acceleration, where the time shortened by LENT-SSE increases as the timeout increases. Setting the rlimit from 1e6 to 10e6, LENT-SSE can also achieve acceleration. The reduction of time of each configuration comes from the reduction of path reachability time, showing the



(a) Total Time (Timeout) (b) Path' Time (Timeout)

(c) Total Time (Rlimit) (d) Path' Time (Rlimit)

Figure 11: Impact of Timeout/Rlimit (RQ4-1)



(a) Total Time (b) Path' Time

Figure 12: Impact of Transaction Depth (RQ4-2)

effectiveness of LENT-SSE. As for the comparison between timeout and rlimit, these two parameters have no significant difference in the test groups of multiple-TX SE.

RQ4-2: What is the impact of the transaction depth?

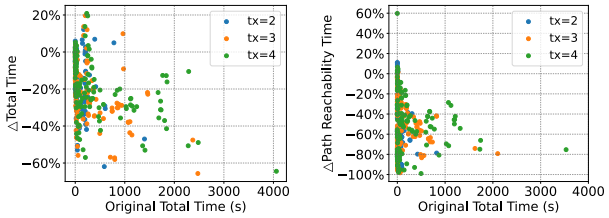
Approach. We run the LENT-Mythril and Mythril in different transaction depths (tx=2,3,4), setting the rlimit as 5,000,000 in order to compare time costs on the SmartBugs dataset.

Results. The results are shown in Figure 12. As the transaction depth increases, the total time and path reachability time increase. Moreover, the reduction of total time in tx=3 (32.22%) and tx=4 (30.57%) is higher than that in tx=2 (22.75%). The absolute value of the reduction of path reachability time also increases as the transaction depth increases. However, the ratio of the reduction of path reachability decreases. The reason might be that, given the same rlimit of the solver, there are more UNKNOWN paths in deeper transactions.

RQ4-3: What is the impact of the contract complexity?

Approach. We measure the complexity of a contract by measuring the total time in its original SE. In this way, we can put the contracts and the reduction of time into scatter diagrams.

Results. Figure 13 shows the scatter diagrams for total time and path reachability time. We observe that most of the points are distributed like a triangle in the bottom left corner position. Besides, the position of the point closer to the right tends to be lower in most cases. This indicates that, in general, the more complex the smart contract, the more time cost reduction LENT-SSE provides.



(a) Δ Total Time (b) Δ Path' Time

Figure 13: Impact of Complexity of Contracts (RQ4-3)

Answer to RQ4. The timeout/rlimit, number of transactions, and complexity of contracts are the key parameters. The higher timeout/rlimit, transaction depth, and contract complexity could lead to better acceleration results of LENT-SSE.

8 Related Work

8.1 SE of Smart Contract

With the development of smart contracts, code analysis tools are needed by the community. Symbolic tools are proposed for smart contracts. Oyente [16] is one of the early proposed tools to detect vulnerabilities in smart contracts by symbolic execution. There are many other symbolic tools based on Oyente (e.g., Maian [20], HoneyBadger [26], and Osiris [25]) that target different vulnerabilities. TeEther [13] is proposed to exploit the contracts by SE. Mythril [18] is proposed to validate EVM smart contracts. Manticore [17] is proposed to achieve SE in both contract codes and binary codes. Since most of these SE tools are open-source, empirical research has been done to evaluate the efficiency of these SE tools in validating smart contracts [6, 8, 21].

Relationship. The above works released symbolic execution tools for smart contracts. And the goal of our work is to accelerate path exploration in symbolic execution. Hence LENT-SSE was applied to the Mythril. The choice of Mythril will be explained in §9. In future work, LENT-SSE can be applied to other tools, such as Manticore and other smart contract symbolic execution tools with similar state-transition models.

8.2 Pruning (Selecting) Paths in SE

There are several studies that focus on how to prune the paths to solve path explosion problem. Note that we also regard some works on selecting important paths as pruning studies. KLEE [4] proposed various optimization mechanisms to reduce the overhead of path exploration, and other studies have reduced the number of paths to be explored from various perspectives, such as problem partitioning and reduction based on ranged analysis [24], non-important code reduction in chopped SE [27], path discarding based on explored paths with the same side effects [2], reducing the number of solves based on program summary trees [22], eliminating paths by post-conditioned approach [30], reducing the path space by merging states [14], eliminating redundant execution of multi-threaded code guided by assertions [9], pruning paths in event-driven GUI programs [10], and path reduction based on program dependencies [29]. As for SE in smart contracts, M-Pro [32] and MTxplorer [31] are proposed to prune the paths by parsing the dependency of states in different transactions.

Difference. These related studies focus on pruning the paths to shorten the time. LENT-SSE does not prune the paths but skips the checking of path reachability of paths. In this way, the time can also be shortened. In our other experiments, setting tx=2, 3, 4, we find the the pruning-based method (M-Pro) saves the time respectively by 30.7%, 22.5%, and 17.9% on SmartBugs dataset. However, we consider that M-Pro [32], and MTxplorer [31] are constructed on different Mythril versions and ideas, thus it would be unfair to directly compare the performance with LENT-SSE. Future work can be done on the combination of the pruning-based method and speculation-based method, in order to achieve better performance.

8.3 Optimization of SMT Solving in SE

Improving the efficiency of constraint solving [5] can help accelerate path exploration. Previous related studies include exploring multiple paths at once by using intermediate solutions [33], prechecking the unsatisfiability of array constraints [23], using the contextual information [15], etc.

Difference. These studies aim at improving SMT solving to enhance SE. LENT-SSE does not spend efforts on the mechanism of SMT solving, but it performs like a middle layer between the symbolic EVM and the SMT solver. It does not improve the speed of solving the constraints in the SMT solver. It reduces the times of triggering the SMT solver.

8.4 Reducing SMT Solving in SE

SSE [34, 35] reduces SMT solving by skipping path reachability checking, and LENT-SSE is a new heuristic for multiple-TX SSE in smart contracts, which have been quantitatively compared in §7.2. There are also several studies that focus on reducing SMT solving by reusing the constraints, such as Green [28] and GreenTrie [11], which are compared in §4.3.

9 Threat to Validity

External Validity: Contracts. The external validity depends on the contracts that are used for statistics. As mentioned in §7, setting tx=2,3,4, there are 4,5,7 complicated contracts that run out of memory. Hence these contracts are not counted in the comparison of the specific configuration. Once we have enough computing resources, the SE of those contracts could be finished, and the effectiveness can be changed. Moreover, as measured in §7, the contracts with higher complexity usually gain better acceleration.

External Validity: Parameter Settings. The external validity also depends on the parameter settings of the experiments. As shown in §7, the parameter settings can affect the results of experiments. However, the parameters (tx, dataset, rlimit, memory) could have infinite compositions that this paper cannot cover. For example, once we use a smaller rlimit or memory as a restriction, LENT-SSE might perform worse.

Internal Validity: Tool. The internal validity depends on the tool that is applied with LENT-SSE. There are three state-of-the-art and widely used (1K+ Github Stars) SE tools in smart contracts: Oyente [16], Mythril[18], and Manticore[17]. Unfortunately, Oyente does not support multiple-TX SE, and Manticore has been claimed on its page that it was no longer internally maintained. Hence, we choose Mythril to apply LENT-SSE in this paper. When LENT-SSE is applied to other tools, the results could be different.

10 Conclusion

This paper proposes a quantitative analysis of the bottleneck and path redundancy of multiple-transaction symbolic execution of smart contracts. Based on the findings, this paper proposes LENT-SSE as a new heuristic for speculative symbolic execution of smart contracts by leveraging the executed and near transactions. The method consists of an executed-transaction-based skipping algorithm and a near-transaction-based recalling algorithm. Experimental results on the third-party dataset show that LENT-SSE can reduce the total time by 37.4% and the solving time of paths by 65.2% on average without reducing the reported bugs.

Acknowledgment

This research is supported by the National Key Research and Development Program of China (2023YFB2704801), the National Natural Science Foundation of China (62332004, 62302538), and the Hong Kong RGC Project (No. PolyU15231223, PolyU15224121).

Appendix

There are two ideas in the previous extended version of SSE [34, 35]: the unsatisfiable core-based backtracking and the absurdity-based optimization. In the evaluation in the paper, we use the unsatisfiable core-based backtracking, but do not use the absurdity based optimization, as follows.

Table 5: Comparison between Different Heuristics in SSE on SmartBugs Dataset

TX'	#Con'	L'	# Shortest Time			Overall Time (s)	
			Origin	SSE1	SSE2	SSE1	SSE2
1	138	2	25	51	62	1503.2	1852.0
		4	21	54	63	1630.6	1950.7
		8	31	90	17	2048.2	3120.5
		16	35	86	17	3974.0	4220.2
2	137	2	93	29	15	14546.0	18869.7
		4	73	49	15	15511.7	18714.9
		8	72	34	31	17085.2	20574.8
		16	74	38	25	21920.7	22778.0

As shown in Table 5, in SSE1, we only use unsatisfiable core based backtracking; in SSE2, we use both unsatisfiable core based backtracking and absurdity based optimization. We count the contracts with the shortest time (best performance) of each type (origin, SSE1, SS2) to compare the performance between SSE1 and SSE2. In most configurations of TX-depth (TX') and SSE-Limit (L'), the SSE1 gets more contracts in shortest time. As for the overall time, the SSE1 is faster than the SSE2 in all test groups. We infer the reason is that, during the SE in the Mythril, once one side is solved to be unreachable, the other side could be also unreachable, since some variables in the subsequent states might be changed or the constraints might exceed the solver's limit. In this case, the absurdity based optimization does not work well in Mythril. However, using other symbolic tools except for Mythril, the performance could be different (e.g., the unsatisfiable core tracking of different SMT solvers might show different performance). Therefore, we actually use the SSE1 as the SSE in the paper to evaluate with LENT-SSE.

References

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *Comput. Surveys* 51, 3 (2018), 1–39.

[2] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking path explosion in constraint-based test generation. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 351–366.

[3] Vitalik Buterin. 2013. *Ethereum white paper*. https://ethereum.org/whitepaper/.

[4] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *USENIX Symposium on Operating Systems Design and Implementation*.

[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

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

[7] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2nd IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain*. 8–15.

[8] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 415–427.

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

[10] Casper S Jensen, Mukul R Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 67–77.

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

[12] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[13] Johannes Krupp and Christian Rossow. 2018. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium*. 1317–1333.

[14] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 193–204.

[15] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan De Halleux. 2010. Flopsy-search-based floating point constraint solving for symbolic execution. In *IFIP International Conference on Testing Software and Systems*. Springer, 142–157.

[16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM CCS*. 254–269.

[17] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *IEEE/ACM International Conference on Automated Software Engineering*.

[18] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. In *9th Annual HITB Security Conference*, Vol. 54.

[19] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.

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

[21] Daniel Perez and Ben Livshits. 2021. Smart contract vulnerabilities: Vulnerable does not imply exploited. In *30th USENIX Security Symposium*.

[22] Rui Qiu, Guowei Yang, Corina S. Pasareanu, and Sarfraz Khurshid. 2015. Compositional symbolic execution with memoized replay. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. 632–642.

[23] Ziqi Shuai, Zhenbang Chen, Yufeng Zhang, Jun Sun, and Ji Wang. 2021. Type and interval aware array constraint solving for symbolic execution. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 361–373.

[24] Junaid Haroon Siddiqui and Sarfraz Khurshid. 2012. Scaling symbolic execution using ranged analysis. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*. 523–536.

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

[26] Christof Ferreira Torres, Mathis Steichen, et al. 2019. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *USENIX Security Symposium*.

[27] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 350–360.

[28] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*.

[29] Haijun Wang, Ting Liu, Xiaohong Guan, Chao Shen, Qinghua Zheng, and Zijiang Yang. 2016. Dependence guided symbolic execution. *IEEE Transactions on Software Engineering (TSE)* 3 (2016), 252–271.

[30] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2017. Eliminating path redundancy via postconditioned symbolic execution. *IEEE Transactions on Software Engineering (TSE)* 1 (2017), 25–43.

[31] 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)*. IEEE, 616–627.

[32] 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 *IEEE 30th International Symposium on Software Reliability Engineering*. 456–462.

[33] Yufeng Zhang, Zhenbang Chen, Ziqi Shuai, Tianqi Zhang, Kenli Li, and Ji Wang. 2020. Multiplex symbolic execution: exploring multiple paths by solving once. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 846–857.

[34] Yufeng Zhang, Zhenbang Chen, and Ji Wang. 2012. S2PF: speculative symbolic PathFinder. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[35] Yufeng Zhang, Zhenbang Chen, and Ji Wang. 2012. Speculative symbolic execution. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. 101–110.

[36] 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*. 740–751.

[37] Peilin Zheng, Zibin Zheng, Jiajing Wu, and Hong-ning Dai. 2020. XBlock-ETH: Extracting and exploring blockchain data from Ethereum. *IEEE Open Journal of the Computer Society* (2020), 95–106.

Received 2024-04-12; accepted 2024-07-03