Efficiently Detecting Reentrancy Vulnerabilities in Complex Smart Contracts

ZEXU WANG, Sun Yat-sen University, China and Peng Cheng Laboratory, China
JIACHI CHEN, Sun Yat-sen University, China
YANLIN WANG, Sun Yat-sen University, China
YU ZHANG, Harbin Institute of Technology, China and Peng Cheng Laboratory, China
WEIZHE ZHANG, Harbin Institute of Technology, China and Peng Cheng Laboratory, China
ZIBIN ZHENG*, Sun Yat-sen University, China and GuangDong Engineering Technology Research Center of Blockchain, China

Reentrancy vulnerability as one of the most notorious vulnerabilities, has been a prominent topic in smart contract security research. Research shows that existing vulnerability detection presents a range of challenges, especially as smart contracts continue to increase in complexity. Existing tools perform poorly in terms of efficiency and successful detection rates for vulnerabilities in complex contracts.

To effectively detect reentrancy vulnerabilities in contracts with complex logic, we propose a tool named SliSE. SliSE's detection process consists of two stages: *Warning Search* and *Symbolic Execution Verification*. In Stage I, SliSE utilizes program slicing to analyze the *Inter-contract Program Dependency Graph* (I-PDG) of the contract, and collects suspicious vulnerability information as warnings. In Stage II, symbolic execution is employed to verify the reachability of these warnings, thereby enhancing vulnerability detection accuracy. SliSE obtained the best performance compared with eight state-of-the-art detection tools. It achieved an F1 score of 78.65%, surpassing the highest score recorded by an existing tool of 9.26%. Additionally, it attained a recall rate exceeding 90% for detection of contracts on Ethereum. Overall, SliSE provides a robust and efficient method for detection of Reentrancy vulnerabilities for complex contracts.

CCS Concepts: • **Software and its engineering → Software verification and validation**; *Software testing and debugging*.

Additional Key Words and Phrases: Reentrancy detection, Program slicing, Symbolic execution

ACM Reference Format:
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. https://doi.org/10.1145/3643734

*Corresponding Author

Authors' addresses: Zexu Wang, Sun Yat-sen University, Zhuhai, China and Peng Cheng Laboratory, Shenzhen, China, wangzx97@mail2.sysu.edu.cn; Jiachi Chen, Sun Yat-sen University, Zhuhai, China, chenjch86@mail.sysu.edu.cn; Yanlin Wang, Sun Yat-sen University, Zhuhai, China, wangylin36@mail.sysu.edu.cn; Yu Zhang, Harbin Institute of Technology, Harbin, China and Peng Cheng Laboratory, Shenzhen, China, yuzhang@hit.edu.cn; Weizhe Zhang, Harbin Institute of Technology, Harbin, China and Peng Cheng Laboratory, Shenzhen, China, wzzhang@hit.edu.cn; Zibin Zheng, Sun Yat-sen University, Zhuhai, China and GuangDong Engineering Technology Research Center of Blockchain, Zhuhai, China, zhzibin@mail.sysu.edu.cn.

ACM 2994-970X/2024/7-ART8
https://doi.org/10.1145/3643734

1 INTRODUCTION

As decentralized applications (DApps) become more versatile in functionality, the complexity of the underlying contract logic has correspondingly increased. This escalation in complexity poses significant challenges for existing tools in detecting vulnerabilities within complex contracts. Reentrancy vulnerabilities are one of the most notorious types [5]. Starting from the DAO Reentrancy attack [11] in 2016 that caused a $150 million loss in digital assets, Reentrancy attacks on blockchain continue to occur. Concurrently, many academic studies and tools for detecting Reentrancy vulnerabilities have emerged.

Various approaches have been employed to detect Reentrancy vulnerabilities, including symbolic execution [9, 25, 27, 36, 42], fuzz testing [8, 21, 28, 32], static analysis [1, 12, 40], and formal verification [10, 12, 37]. However, most existing tools have been evaluated on relatively simple contract datasets, e.g., the *SmartBugs Dataset* [31], lacking experimental assessments on real-world complex contracts. To ascertain their efficacy in complex DApps, Zheng et al. [45] curated a dataset that comprises 895 vulnerabilities. These vulnerabilities were obtained from 1,322 open-source DApp audit reports provided by 30 blockchain security companies, covering 25 types of vulnerabilities. Compared to the *SmartBugs dataset* [31], it has approximately 25 times the average lines of code and 30 times the average function count. The study also evaluated five state-of-the-art vulnerability detection tools [1, 8, 9, 12, 37] on this dataset. The experimental results revealed that most tools had a low success detection rate (less than 30%), especially for Reentrancy vulnerabilities (less than 11%). This emphasizes the necessity of focusing on detecting real-world vulnerabilities in complex contracts rather than simple toy contracts in future research.

Real-world Reentrancy attack events usually involve complex function call relationships. However, most existing tools focus only on the security of a single function within the contract, which is insufficient to guarantee the overall contract safety. Compositional Reentrancy vulnerabilities of smart contracts are introduced by cross-contract interactions, such as cross-function Reentrancy [34] and cross-contract Reentrancy [35], presenting considerable challenges. Effective detection of compositional Reentrancy vulnerabilities requires a thorough analysis of contract interactions and a systematic examination of data and control flow transitions. Incorporating program semantics could enhance the accuracy of identifying critical data and control flows, thereby improving compositional Reentrancy vulnerability detection effectiveness. Additionally, the computation of dynamic jump addresses poses another major challenge, often resulting in incomplete Control Flow Graph (CFG) paths. Despite complex function call relationships may generate numerous execution paths, many current tools predominantly rely on static stack simulation for path recovery. This method struggles with dynamic jump addresses, leaving CFG paths incomplete and compromising the accuracy of symbolic execution. Therefore, the analysis of compositional Reentrancy vulnerability and CFG path recovery are the key challenges in complex contract Reentrancy vulnerability detection.

To address these challenges, we propose the SliSE method, which combines symbolic execution with program slicing to detect Reentrancy vulnerabilities in complex contracts. The detection process is divided into two stages: *Warning Search* and *Symbolic Execution Verification*. In the *Warning Search* stage, program slicing is performed based on program dependencies to search and extract critical paths. Subsequently, in the *Symbolic Execution Verification* stage, the reachability of these critical paths is further verified to achieve efficient Reentrancy vulnerability detection. Through program dependency analysis, the I-PDG (Inter-contract Program Dependency Graph) of contract is constructed, which can provide cross-contract data and control dependencies. Combined with our Reentrancy vulnerability slicing standards, the I-PDG is sliced to prune paths. SliSE analyzes program dependency among critical instruction statements (such as *require*, *assert*, etc.)

to model the semantics, then identifies data and control flow transitions. With this information, it verifies whether these semantics adhere to the secure C-E-I (*Check->Effect->Interaction*) patten. This pattern mandates timely state updates before interacting with external contracts, maintaining atomicity of transactions and prevent Reentrancy (see section 2 for details). If deviations from this pattern are detected, SliSE generates a warning that includes the corresponding function and location information. In the *Symbolic Execution Verification* stage, SliSE employs the CFG path recovery algorithm to recovery execution paths and gather essential path constraints. Finally, it proves the reachability of the path and the existence of vulnerabilities through constraint solving.

We evaluated SliSE's performance by comparing the detection results with eight state-of-the-art tools, analyzing its performance in detecting Reentrancy vulnerabilities from complex contracts. The results show that SliSE performs well in detecting complex contract Reentrancy vulnerabilities, with the F1 score of 78.65%, significantly exceeding the highest score of 9.26% achieved by the eight state-of-the-art tools. To assess its effectiveness in detecting Reentrancy vulnerabilities on Ethereum, we used two publicly available datasets [31, 46], SliSE demonstrated an outstanding recall rate of 92.68% and maintained the highest F1 scores compared to the existing tools. Through ablation experiments for each stage, we confirmed the critical importance of each stage in the overall process. Notably, precise path pruning during stage I is instrumental for efficient vulnerability detection. This path pruning process contributed to a significant increase in the F1 score, elevating it from 6.59% to 78.65%. Additionally, with the stage II of symbolic execution verification, we observed a substantial reduction in false positives, resulting in a precision improvement from 47.30% to 72.16%. Overall, SliSE provides a robust and efficient method to efficiently detect Reentrancy vulnerabilities in complex contracts.

The main contributions of our work are as follows:

(1) We propose an approach that combines program slicing and symbolic execution to efficiently detect Reentrancy vulnerabilities within complex contracts. By pruning and verifying the reachability of critical paths, we achieve effective detection.
(2) We design the SliSE tool, which efficiently detects complex contracts Reentrancy vulnerabilities. Comparative experiments with state-of-the-art tools substantiate its effectiveness and efficiency.
(3) We have made our SliSE tool's source code and experimental dataset publicly available at https://github.com/SliSE-SC/SliSE.

The paper is organized as follows. In Section 2, we provide essential background and highlight challenges in detecting complex contract Reentrancy vulnerabilities through motivating examples. Section 3 outlines the workflow and technical details of SliSE. We evaluate the performance and efficiency of SliSE in Section 4, while Section 5 discusses existing tool capabilities in complex contract Reentrancy vulnerability detection and threats analysis. Section 6 summarizes related work, and Section 7 concludes this paper.

2 BACKGROUND AND MOTIVATION

2.1 Reentrancy Vulnerability Detection

C-E-I (*Check->Effect->Interaction*) is a critical security programming pattern in smart contracts. It requires that external interactions should only occur after all preconditions are checked and internal state updates are effected [14, 19]. For example, when transferring tokens to an external contract (***Interaction phase***), it is essential to perform user balance checks (***Check phase***) and update balances (***Effect phase***) first. In case of an attacker attempting Reentrancy, the transaction will revert due to the balance check failure in ***Check phase***, effectively preventing Reentrancy. Evaluating

whether token transfers in the contract adhere to the secure C-E-I (*Check->Effect->Interaction*) pattern is pivotal in Reentrancy vulnerability detection.

Reentrancy vulnerabilities typically stem from the violation of the secure *Check->Effect->Interaction* (C-E-I) pattern, with attackers exploiting the smart contract's *fallback mechanism*. The *Check->Effect->Interaction* (C-E-I) security pattern necessitates that state changes occur before contract interactions, ensuring timely state updates. Smart contracts' *fallback mechanism* activates automatically upon receiving Ether (native tokens). Some token standards, such as ERC-777 [20] and ERC-1155 [38], emulate the native token's *fallback mechanism* using *hook functions*. If the state is not promptly updated, transferring tokens to the attacker can trigger their fallback function, enabling them to hijack the logic and exploit Reentrancy vulnerabilities. This disrupts the atomicity of the transaction, leading to multiple executions from a single function call.

Figure 1 displays a Reentrancy vulnerability. The *withdraw* function allows users to transfer a amount of tokens. In L5, it first verifies whether the user's balance (*userBalance*) exceeds the transferred amount (***Check phase***). If the condition is met, execution proceeds, otherwise, the transaction is reverted. L6 facilitates the transfer of *_amount* tokens to *msg.sender* (***Interaction phase***). Following this, L7 updates the balance of *msg.sender* (***Effect Phase***). This sequence in the *withdraw* function violates the secure *Check->Effect->Interaction* (C-E-I) transfer pattern, which undermines transaction atomicity and triggers Reentrancy attacks. This vulnerability arises in L6, where a transfer is initiated by *msg.sender* (external user) using *call.value*. Since the deduction of the attacker's balance is deferred to L7, the balance check (***Check phase***) in L5 remains satisfied, facilitating further token transfers to the attacker. Exploiting this, an attacker can repeatedly trigger the *withdraw* function through the *fallback mechanism*. To rectify this, the transfer logic should follow the *Check->Effect->Interaction* (C-E-I) patten, with the execution sequence as *L5->L7->L6*. The balance update (L7) must precede the transfer (L6). This arrangement ensures that even if the attacker reenters the *withdraw* function via the *fallback mechanism*, the reentry fails due to an unsatisfied balance check condition (L5).

```
1  contract ContractA {
2    ...
3    function withdraw(address _contractB, uint _amount) public {
4        userBalance = _contractB.getBalance(msg.sender);
5        require(userBalance >= _amount);
6        msg.sender.call{value:_amount}("");
7        _contractB.reduceBalance(msg.sender,_amount);
8    }
9  }
10 contract ContractB {
11   mapping(address => uint) balances;
12   ...
13   function getBalance(address _address) public view returns (uint) {
14       return balances[_address];
15   }
16
17   function reduceBalance(address _address, uint amount) public {
18       balances[_address] = balances[_address] - amount;
19   }
20 }
```

Fig. 1. The example of Reentrancy

Compared to native token (ETH), ERC tokens utilize *hook functions* to simulate the native token's *fallback mechanism*. According to different implementation methods of *fallback mechanism*, Reentrancy vulnerabilities are divided into the following two categories:

- **Reentrancy with ETH:** Transferring ETH, using the native *fallback mechanism* to implement callback;
- **Reentrancy with ERC Token:** Transferring ERC tokens (derivative tokens), the *hook function* is used to implement callback.

Existing *hook function* examples, such as ERC-777 and ERC-1155, are designed to address the problem of asset lock-in at receiving addresses. ERC-777 tokens require invoking the *ERC777TokensSender* (a hook function) before state updates [20]. If these updates are not synchronized, transaction atomicity can be compromised, resulting in Reentrancy vulnerabilities. Figure 2 illustrates a cross-contract Reentrancy incident on Cream Finance, where the attack occurred on August 30, 2021. The attacker exploited the ERC-777 token's *hook function* to borrow digital currencies twice, even though they had pledged assets only once. In the critical execution logic of the *borrowFresh* function, calls *borrowAllowed* to check user status (***Check phase***) in L11, *ERCToken(borrower).transfer(...)* executes the REC-777 token transfer (***Interaction phase***) in L13, and updates the user's balance (***Effect phase***) in L15. This violation of the *Check->Effect->Interaction* (C-E-I) transfer pattern resulted in Cream Finance losing approximately $18.8 million worth of digital assets. Notably, even though the *borrowInternal* function in L5 had a Reentrancy lock (*nonReentrant*), it still fell prey to a Reentrancy attack. **This underscores that a single function's Reentrancy lock cannot prevent cross-contract Reentrancy attacks.** The execution of *borrow* function in L2 involves multiple internal function calls, each with distinct purposes. It is essential to extract critical semantic features to ascertain whether the contract abides by the secure *Check->Effect->Interaction* (C-E-I) pattern. This emphasizes the significance of compositional security analysis when identifying Reentrancy vulnerabilities in complex contracts. Beyond examining the Reentrancy of individual functions, it is crucial to consider insecure implementations within the contract semantics.

2.2 Motivation Examples and Challenges

2.2.1 Semantic Modeling of Complex Contracts. The complexity of function call relationships complicates semantic analysis. Figure 3 depicts the key function calls during the execution of the *borrow* function in Figure 2. This figure highlights the intricate logic embedded within this function's implementation. Due to the modular design of smart contract functions, there's an inherent complexity in function calls. Through semantic analysis, we discern that the overall token transfer process violates the *Check->Effect->Interaction* (C-E-I) pattern. Crucially, the ***Interaction phase*** is executed before the ***Effect phase***, and this sequence is a pivotal factor for triggering Reentrancy. When the state is not updated promptly, the condition checked in the ***Check phase*** becomes invalid. Simultaneously, external contract callbacks (attacker) intercept the execution, introducing malicious code for Reentrancy. For example, in Cream Finance's case, token transfers using the ERC-777 standard allow the receivers (attacker) to execute specific logic from the hook function upon receiving ERC-777 tokens, which is an important reason for Reentrancy attacks.

Motivation: Existing approaches usually lack comprehensive compositional analysis of smart contract program semantics, typically narrowing their focus to individual functions. This limitation may result in missed vulnerabilities and false positives in detection results. As shown in Figure 3, the execution of the *borrow* function involves complex cross-contract interactions and function calls. Tools such as Sailfish [1] and Mythril [9] struggle to analyze data flow during these cross-contract interactions, leading to missed vulnerability detection. On the other hand, static analysis tools like Slither [12] primarily ensure individual function security, neglecting compositional security

```
contract CreamFinance_Reentrancy{
  function borrow(uint borrowAmount) returns (uint) {
      return borrowInternal(borrowAmount);
  }
  function borrowInternal(uint borrowAmount) internal nonReentrant returns
       (uint) {
      uint error = accrueInterest();
      ...
      return borrowFresh(msg.sender, borrowAmount);
  }
  function borrowFresh(address payable borrower, uint borrowAmount)
      internal returns (uint) {
      uint allowed = comptroller.borrowAllowed(address(this), borrower,
          borrowAmount);
      ...
      ERCToken(borrower).transfer(borrowAmount);
      ...
      comptroller.borrowVerify(address(this), borrower, borrowAmount);
      return uint(Error.NO_ERROR);
  }
}
```

Fig. 2. The *borrow* function causing Cream Finance's Reentrancy

analysis. This limitation stems from their inability to systematically analyze state interdependencies among multiple functions, leading to a significant number of false positives in the detection results. **Challenge:** The primary challenge in semantic modeling lies in comprehensively analyzing program dependencies and precisely capturing data dependencies during contract interactions.

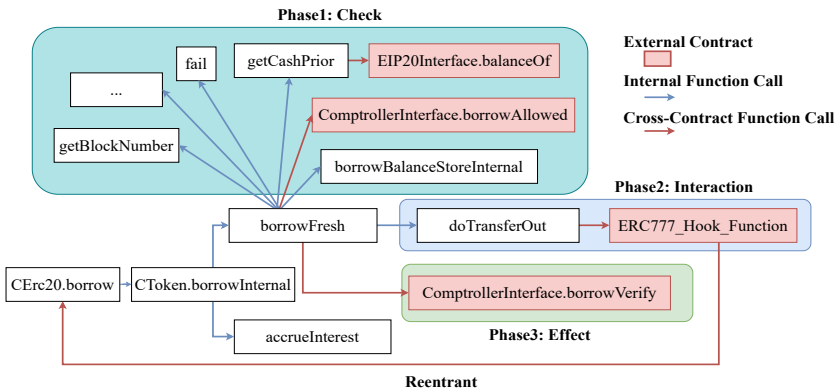

Fig. 3. Execution phases and function calls of the *borrow* function

2.2.2 CFG Path Recovery. Accurately collecting and propagating path constraints are crucial for achieving effective cross-contract symbolic execution, directly impacting vulnerability detection precision. In Figure 4, the essential CFG paths of the *borrow* function in Figure 2 are depicted. The blue arrow represents the dynamic jump edge (*Orphan Jump*[1]) [10], where the jump address of

[1] *Orphan Jump* lacks a preceding *PUSH* opcode, making its jump target address challenging to compute immediately.

the *JUMP* opcode cannot be determined through static analysis. The blue oval block represents *Orphan Jump* address block, while red filled squares represent blocks containing cross-contract interaction. As shown in Figure 4, the execution process of the *borrow* function involves numerous dynamic jump edge calculations and cross-contract interactions. Existing tools such as Mythril [9], Sailfish [1], and Manticore [27] encounter challenges due to difficulties in computing dynamic jump addresses and a lack of support for cross-contract analysis. Consequently, their CFG paths are incomplete, resulting in issues such as missing path constraints and incomplete path traversal. This leads to a significant number of false negatives in the detection results.

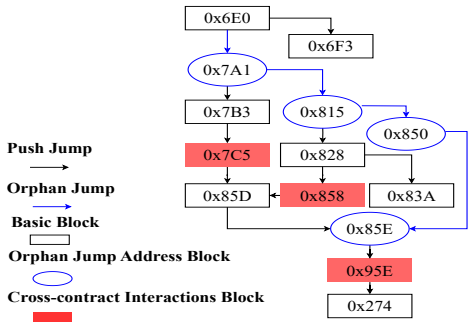

Fig. 4. Control Flow Graph (CFG) of the *borrow* function.

Motivation: Recovering dynamic jump edges is crucial for collecting path constraints and cross-contract analysis. Although traditional symbolic execution proves beneficial in CFG path recovery, it is still challenging in dynamic jump edge calculations. This limitation hinders their capability to detect vulnerabilities in complex contracts. While some tools employ static stack emulation for CFG recovery, they struggle with dynamic jump edges, leaving contract CFG paths incomplete. Furthermore, effective cross-contract analysis on complete CFG paths is essential for efficient symbolic execution. During cross-contract interactions, context switching ensures smooth collection and propagation of path constraints. However, state-of-the-art symbolic execution tools like Sailfish [1] and Mythril [9] frequently face difficulties in cross-contract interactions, primarily due to path losses, leading to timeouts and exacerbated path explosion.
Challenge: Efficiently determining dynamic jump addresses and recovering the CFG path present significant challenges.

3 METHODOLOGY

In this section, we introduce the workflow and delve into the technical details of SliSE.

3.1 Overview

The SliSE method efficiently detects complex contract Reentrancy vulnerabilities by combining program slicing with symbolic execution verification. The process, as illustrated in Figure 5, takes source code as input, reports the presence of vulnerabilities and their corresponding locations. It consists of two main stages: *Warning Search* and *Symbolic Execution Verification*. In Stage I, SliSE analyzes program dependencies through the Abstract Syntax Tree (AST) to construct the Inter-contract Program Dependency Graph (I-PDG) of the contract. It then performs slicing analysis based on Reentrancy vulnerability characteristics to identify suspicious vulnerabilities' functions and locations, which are subsequently reported as warning information. In Stage II, critical paths containing warning information are extracted, followed by symbolic execution to validate the

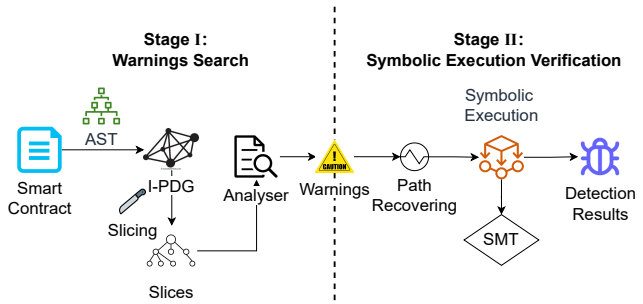

Fig. 5. The workflow of SliSE

reachability of these paths. This combined static analysis and symbolic execution to effectively detect Reentrancy vulnerabilities in complex contracts.

3.2 Stage I: Warnings Search

To improve search efficiency, SliSE performs program slicing analysis on the global program dependency of contracts, pruning irrelevant paths to enhance detection efficiency. Initially, SliSE compiles the contract source code to obtain the corresponding Abstract Syntax Tree (AST) information. By analyzing the AST, it constructs the *Inter-contract Program Dependency Graph* (I-PDG) for the contract. Combined with the characteristics of Reentrancy vulnerabilities, program slicing is then applied to the I-PDG. This process involves verifying whether the corresponding code semantics adhere to the secure *Check->Effect->Interaction* (C-E-I) pattern. The results of this analysis yield warning information about Reentrancy vulnerabilities, which serves as input for Stage II.

Algorithm 1: Constructing I-PDG

Input: Inter-contract Control Flow Graph (I-CFG)
Output: Inter-contract Program Dependency Graph (I-PDG)

```
Function Constructing_I-PDG(I-CFG):
    I-PDG ← Initialize an empty graph;
    foreach node in I-CFG do
        I-PDG.add_node(node);
        foreach successor in node.successors do
            I-PDG.add_node(successor);
            I-PDG.add_control_edge(node, successor);
            if modify(node, successor.variable) then
                I-PDG.add_data_edge(node, successor);
            end
        end
    end
    return I-PDG;
```

3.2.1 Constructing Inter-contract Program Dependency Graph. For a comprehensive analysis of smart contract program semantics, we construct the *Inter-contract Program Dependency Graph* (I-PDG) by analyzing the overall dependency relationships within the contract's AST. While the existing *Inter-contract Control Flow Graph* (I-CFG) [26] transforms cross-contract calls into global jumps between statement blocks, offering an overview of the global control flow, it faces challenges

in achieving a comprehensive global data flow analysis due to intricate execution paths and frequent cross-contract interactions. Building upon the contract's I-CFG, SliSE performs program dependency analysis between each statement block through the AST to create the *Inter-Contract Program Dependency Graph* (I-PDG). In this I-PDG, nodes represent fundamental statements, and edges signify program dependency relationships involving both control and data dependencies. Algorithm 1 outlines the construction process of the I-PDG. In this algorithm, L3–L4 iterate through each node in the I-CFG and add it to the I-PDG, while L5–L11 add the succeeding node of each to the I-PDG. Simultaneously, we analyze relationships between data definitions and usage, as well as control relationships between nodes, introducing data dependency edges and control dependency edges.

Figure 6 outlines the process of constructing the *Inter-contract Program Dependency Graph* (I-PDG) for contracts in Figure 1. Utilizing the foundational nodes from the I-CFG [26] as a starting point, we generate subgraphs of *Control Dependency Graph* (CDG) and *Data Dependency Graph* (DDG) based on the control dependency relationships and the data dependency relationships between the respective nodes. Our main contribution lies in achieving a comprehensive program dependency analysis for cross-contract scenarios. Global program dependencies offer richer information for analyzing global variables, user input, data flow during cross-contract interactions, and more. As shown in Figure 6, the blue solid line represents the existing *Inter-contract Call Dependencies*, enabling direct data flow analysis in cross-contract scenarios. This helps us understand the impact of functions invoked across contracts on global state variables and how this influence spreads.

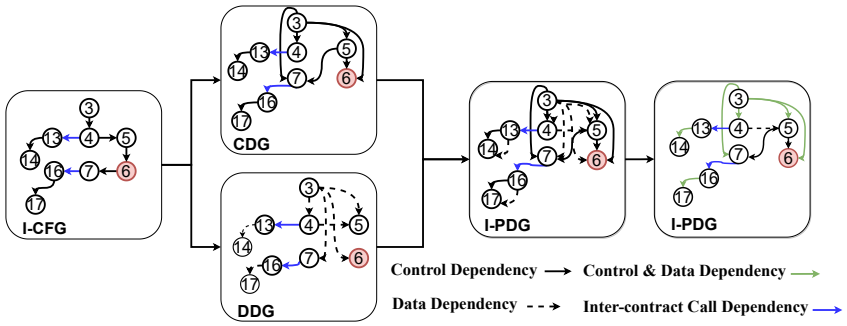

Fig. 6. I-PDG construction for example in Figure 1

3.2.2 Slicing & Analysing. To prune irrelevant paths, we have devised specific slicing criteria based on the unique attributes of these vulnerabilities. These criteria enable precise analysis by extracting essential code sections that exhibit Reentrancy vulnerability traits. Leveraging the global program dependency, we explore relationships among cross-contract call addresses, input variables, and data flows during cross-contract interactions. These relationships are crucial for a comprehensive Reentrancy vulnerability analysis. However, since these dependency relationships are not exclusive to smart contracts, uninformed analysis could lead to significant false positives. To address this, we propose vulnerability slicing rules aligned with Reentrancy attack characteristics. These rules streamline the analysis of code relevant to Reentrancy vulnerabilities, enhancing detection efficiency. Furthermore, by considering the characteristics of Reentrancy vulnerabilities and the *fallback mechanism*, we establish slicing criteria specific to two types: ETH and ERC tokens. These criteria efficiently target and analyze code relevant to Reentrancy vulnerabilities.

- **Rule for Reentrancy with ETH:**
  Backward_Slicing[User_Input_Address.call.value()]

- **Rule for Reentrancy with ERC Token:**
  Backward_Slicing[ERC(User_Input_Address).call_function()]

Rule for Reentrancy with ETH used for vulnerabilities originating from ETH transfers, primarily utilizing the *call.value()*. Consequently, our focus is on the dynamic account address (user input used as address) triggering the *call.value()* function. Employing backward slicing on the I-PDG, we isolate nodes with dependencies, concentrating on sections pertinent to the Reentrancy vulnerability.

Rule for Reentrancy with ERC Token focuses on Reentrancy originating from ERC token transfers. It centers on the function call triggered by the dynamic address contract (user input used as contract address), serving as the entry point. By conducting backward slicing of the I-PDG, we retain nodes with dependencies in the slice. For example, in Figure 2, L13 serves as the slicing entry point. This line calls the function from the dynamic address contract (ERCToken(borrower)). Given the uncertain logic of the external contract, attackers can exploit it by injecting malicious code to achieve Reentrancy.

As depicted in Figure 7, this demonstration follows the *Rule for Reentrancy with ETH* standard for the code in Figure 1. In this example, *msg.sender.call.value* in L6 meets the criteria for slicing, serving as the entry point for conducting a backward slice. The resulting sliced code snippet is displayed on the right side. L3, L4, and L5 contain the code within the backward slice, while L13 and L14 are retained in the slice due to their inter-contract call dependencies. This retention preserves the integrity of dependency relationships. Our Reentrancy vulnerability program slicing criterion facilitates focused analysis of critical code segments while preserving relevant dependency relationships. This approach enables efficient and precise code analysis, mitigating the impact of unrelated code on vulnerability detection's effectiveness and accuracy.

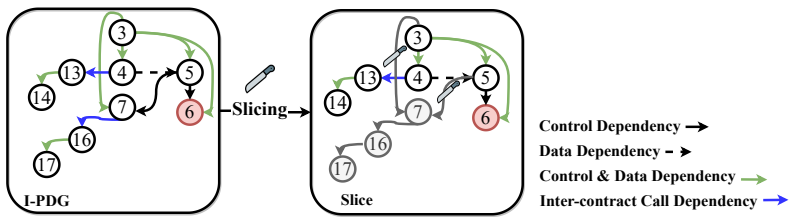

Fig. 7. Slicing process for example in Figure 1

To model and analyze semantics, we scrutinize the program dependencies of critical instruction-related statements. For instance, in Figure 1, we can extract the ***Check phase*** within the code using conditional checks like *require* and *assert*, along with control dependency analysis. The presence of cross-contract function calls indicates the ***Interaction phase***, while statements related to variable updates in the ***Check phase*** correspond to the ***Effect phase***. SliSE employs this information to slice the contract's I-PDG, assessing its adherence to the secure development pattern *Check->Effect->Interaction* (C-E-I), and generating pertinent warning messages.

Figure 8 illustrates the analysis procedure of the slice corresponding to the code presented in Figure 1. Within the slice code, the *require()* in L5 represents the ***Check phase***, involving the *balances[user]* variable. Additionally, *msg.sender.call.value()* in L6 signifies the ***Interaction phase***, aligning with the slicing rule of the *Rule for Reentrancy with ETH*. It is noteworthy that no actions are taken to alter the *balances[user]* variable (modification occurs in L7), indicating that the ***Effect phase*** follows the ***Interaction phase***. This observation plays a crucial role in identifying the Reentrancy vulnerability. When an attacker reenters through the external contract, the state of the ***Check phase*** remains unchanged (*balances[user]* does not decrease). Consequently, it can be

identified as a Reentrancy vulnerability. We compile the location details of the vulnerable function to generate warning information.

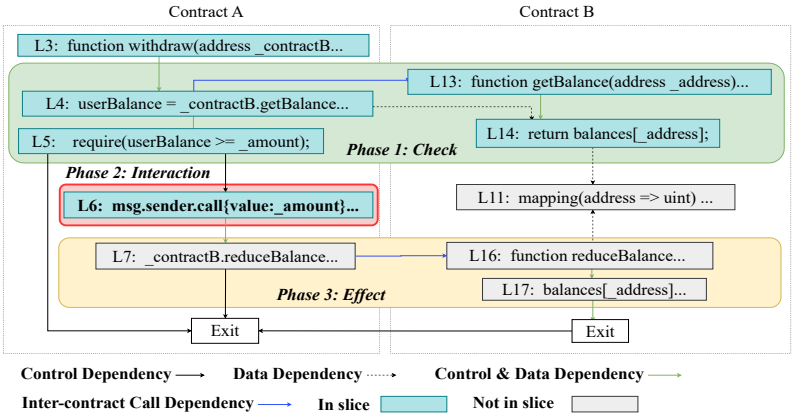

Fig. 8. Analysis of Program Slicing for Figure 1

3.3 Stage II: Symbolic Execution Verification

To reduce false positives and ensure reliable vulnerability detection, SliSE employs symbolic execution for path reachability verification. Initially, SliSE addresses the challenge of calculating dynamic jump addresses (*Orphan Jump*) by using Algorithm 2 combined with SSA[2] (Static Single Assignment) to ensure complete CFG paths. Based on the warning information from Stage I, critical paths are traversed, and path constraints are collected. These constraints are stored in the *Symbolic Register* for later access and retrieval. They are then validated using the *Z3-solver* to confirm path reachability and, consequently, the existence of vulnerabilities.

3.3.1 Path Recovering. To efficiently recover CFG paths, we utilize *constant propagation analysis* combined with SSA to determine the target addresses of *Orphan Jumps*. Initially, the bytecode is divided into multiple blocks, and some blocks connection are recovered through static stack emulation. However, static stack emulation can only recover jump edges for *Push Jumps*[3], and it cannot ascertain the target of an *Orphan Jump* within the current block. This limitation results in the inability to recover the edges of *Orphan Jumps*, leaving the CFG incomplete.

Figure 9 illustrates an example of recovering the CFG with SSA. In the incomplete CFG obtained through static stack emulation, a *JUMP* instruction exists within Block_ID 6, but this block lacks successor blocks, creating what is known as an *Orphan Jump*. The inherent limitation of static stack emulation in addressing the *Orphan Jump* issue lies in its incomplete analysis of variable transitions within the current block. It cannot effectively analyze the flow of values along the execution path of the current block, making it incapable of solving the *Orphan Jump* problem effectively. By leveraging the property of SSA, which ensures that each variable is assigned only once, we can perform global value propagation analysis along the execution path of the *Jump* block. The SSA representation clearly highlights the relationships between variable definitions and their uses within each block. If, prior to the *Jump* instruction, there exists a variable that is defined but

[2]SSA is a property of intermediate languages that mandates each variable to be assigned only once, enabling features like constant propagation analysis.
[3]*Push Jump* is immediately preceded by a *PUSH* opcode, making its jump target address easily calculable.

not used, that variable represents the *stacktop value* and serves as the target of the *Orphan Jump*. If no *stacktop value* is found in the current block, the search continues in the preceding blocks. For instance, in Block_ID 6, where a *JUMP* instruction is present, the *stacktop value* can be located in its predecessor block (Block_ID 3). This approach ensures the precise recovery of target addresses for *Orphan Jumps*, obtaining a complete CFG.

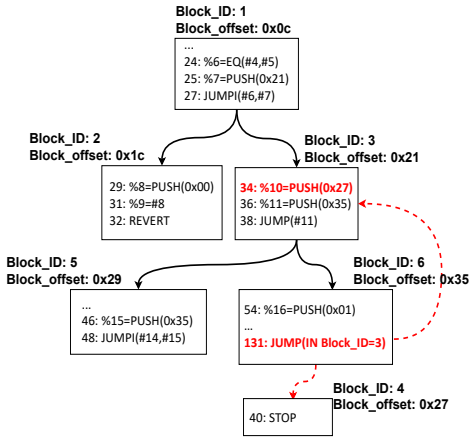

Fig. 9. Recovering the CFG with SSA

To search the target addresses, we utilize constant propagation analysis with SSA to calculate the dynamic target addresses of *Orphan Jumps*. Algorithm 2 outlines the process for CFG path recovery. This process involves transforming the bytecode within each block into SSA form, ensuring that each numeric value is assigned only once. As we traverse the blocks of the incomplete CFG, we update the SSA global variables (*ssaVariables*) based on operations that involve variable assignment or access (e.g., *PUSH, DUP, POP, SWAP*) [39]. For example, in L8–L9, we identify the jump target by searching for unused variables in the current block. Unused variables refer to those defined in the block but are not utilized. If such a variable exists within the current block, it becomes the jump target. However, if it does not, we perform a search within the preceding blocks along the CFG path where the current block is located. The *findUnusedVar* function facilitates this iterative search process. This algorithm ensures accurate propagation of values along the execution path, facilitating the precise recovery of target addresses for *Orphan Jumps* embedded within the CFG.

3.3.2 Symbolically Verifying Path Feasibility. To ensure the accuracy of Reentrancy vulnerability detection, we perform a reachability analysis of the warning path information. While Stage I conducts program slicing analysis to generate warning information related to violating the *Check->Effect->Interaction* (C-E-I) pattern, it lacks verification of the reachability of the warning paths. Specifically, contract *Path Protective Techniques* (PPTs) [40], such as mutex locks and permission checks, can limit the occurrence of Reentrancy vulnerabilities. Static analysis tools often struggle to accurately identify these protective techniques, leading to numerous false positives. To address this, SliSE collects warning path constraints and validates their reachability. When it encounters assignments or accesses involving symbolic values, it uses the *symbolic register* for symbolic state manipulation. Considering the conditions for Reentrancy vulnerability and the characteristics of smart contracts, it collects constraints and sends them to *Z3-solver* for constraint calculation to prove path reachability, significantly enhancing the accuracy of vulnerability detection.

To ensure the integrity of path constraints, we manage symbolic expressions in a key-value pair format for further analysis. By examining relevant operations related to state storage and utilizing Z3 library functions, we store symbolic expressions as key-value pairs in the *Symbolic Register*. The access and storage of symbolic states are crucial for collecting path constraints. By querying the *Symbolic Register*, SliSE determines whether the contract address involved in external contract function calls is a symbolic value, assessing whether the logic of the called contract is controlled by external users. This also facilitates quick assessment of whether a contract might be susceptible to hijacking by malicious callback functions, streamlining the efficient propagation analysis of cross-contract path constraints in various contract contexts during symbolic execution verification.

Algorithm 2: Recovery CFG

```
Input: CFG
Output: Reconstructed CFG
Function RecoveryCFG(CFG):
    foreach block in CFG.blocks do
        jumpTarget ← findUnusedVar (block);
        CFG.add_edge(block, blockAt(jumpTarget));
    end
Function findUnusedVar(block):
    foreach preBlock in block.predecessors do
        StackTopValue ← getUnusedVariables (preBlock);
        return StackTopValue;
    end
```

4 EVALUATION

In this section, we evaluated SliSE's performance by comparing the detection results with eight state-of-the-art tools and analyzing the impact of each stage on the overall detection process. We will address the following research questions:

RQ1. How effective is SliSE in detecting complex contracts Reentrancy vulnerabilities?
RQ2. How effective is SliSE in detecting Reentrancy vulnerabilities on Ethereum?
RQ3. What is the impact of pruning in Stage I?
RQ4. What is the impact of symbolic execution verification in Stage II?

4.1 Experimental Setup

The experiments were conducted on a computer running Ubuntu 18.04.1 LTS, equipped with a 16-core Intel(R) Xeon(R) Gold 5217 processor. We set up the experiments by downloading images or manually configuring. We used default parameters and a time budget of 300 seconds to ensure that the experiments did not excessively consume time, aligning with the approach suggested in [46].

Tools selection following with [45], the experiment using a set of analysis tools, including Slither [12], Mythril [9], Securify [37], Smartian [8], and Sailfish [1]. Additionally, to facilitate a comparative analysis of the performance of existing symbolic execution detection tools, we also integrated three symbolic execution tools, i.e., Oyente [25], Osiris [36], and Manticore [27].

The experimental datasets include DB1, DB2, and DB3. DB1 is sourced from open-source DApp auditing projects, representing off-chain versions with complex contract logic. DB2 comprises real on-chain contracts, most of which have been labeled as positive by existing tools, posing a significant detection challenge. DB3 is the simplest and most widely utilized dataset. Smart contract

of these datasets all originate from real-world production environments, compared and analyzed through *Precision* ($\frac{TP}{TP+FP}$), *Recall* ($\frac{TP}{TP+FN}$), and *F1 score* ($\frac{2*precision*recall}{precision+recall}$). DB1 was employed to answer RQ1, RQ3, and RQ4, while DB2 and DB3 were utilized to answer RQ2, RQ3, and RQ4.

DB1. *Complex Contract Dataset* [45]. Zheng et al. compiled a complex contract dataset that encompasses 895 vulnerabilities from 1,322 open-source DApp audit projects provided by 30 blockchain security companies, with a total of 81 positive labels for Reentrancy vulnerabilities.

DB2. *Ethereum Contract Dataset* [46]. Zheng et al. used existing detection tools to analyze 230,548 verified contracts from Etherscan and obtained this dataset through manual inspection. The dataset includes 21,212 contracts identified as positive Reentrancy vulnerabilities using six Reentrancy detection tools, with 41 of contracts manually verified as true positives.

DB3. *SmartBugs Dataset* [31]. The SmartBugs includes 143 contracts, of which 31 are labeled with Reentrancy vulnerabilities.

Research indicates that manually labeling smart contracts is highly error-prone [46], and achieving precise labeling for all Ethereum smart contracts is a formidable challenge. To ensure the accuracy of our experimental analysis, we conducted RQ2 experiments using the DB2 and DB3 datasets. These datasets have has been presented at top software engineering conferences and are widely recognized within the academic community, serving as a reliable source of ground truth for Ethereum contracts. We have compared key attributes among these datasets, as depicted in Table 1. Notably, the average lines of code (Loc) in DB1 are approximately 24.9 times greater than in DB3 and 3.4 times greater than in DB2. Similarly, the average number of functions is 29.9 times higher in DB3 and 3.6 times higher in DB2 compared to DB1. These findings underscore the high complexity of DB1 dataset from the real production environment, and we will delve into a quantitative analysis of DB1's complexity in Section 5.

Table 1. Statistics for different datasets

Dataset	Loc	# of Subcontracts	# of Functions	# P	# N
DB1	1812.5	12	197.4	81	814
DB2	534.5	6.0	54.2	41	21171
DB3	72.8	1.5	6.6	31	112

4.2 Effectiveness of Detecting Reentrancy for Complex Contracts

Table 2. Statistics of Detection Results by Different Tools

Tool	Mythril			Securify			Slither			Oyente			Osiris			Manticore			Smartian			Sailfish			Ours		
Dataset	DB1	DB2	DB3	DB1	DB2	DB3	DB1	DB2	DB3	DB1	DB2	DB3	DB1	DB2	DB3	DB1	DB2	DB3	DB1	DB2	DB3	DB1	DB2	DB3	DB1	DB2	DB3
# TP	5	8	13	0	31	29	8	36	30	0	32	28	0	34	29	0	0	0	0	7	19	0	26	25	70	38	30
# FP	22	15492	51	0	2356	44	140	18346	67	0	481	39	1	476	31	0	22	0	0	15	9	2	2270	31	27	31	14
# FN	76	33	18	81	10	2	73	5	1	81	9	3	81	7	2	81	41	31	81	34	12	81	15	6	11	3	1
# TN	792	5679	61	814	18815	68	676	2825	45	814	20690	73	813	20695	81	814	21149	112	814	21156	103	812	18901	81	787	21140	98
P	18.52%	0.05%	20.31%	0.00%	1.30%	39.73%	5.41%	0.20%	30.93%	0.00%	6.24%	41.79%	0.00%	6.67%	48.33%	0.00%	0.00%	0.00%	0.00%	31.82%	67.86%	0.00%	1.13%	44.64%	72.16%	55.07%	68.18%
R	6.17%	19.51%	41.94%	0.00%	75.61%	93.55%	9.88%	87.80%	96.77%	0.00%	78.05%	90.32%	0.00%	82.93%	93.55%	0.00%	0.00%	0.00%	0.00%	17.07%	61.29%	0.00%	63.41%	80.65%	86.42%	92.68%	96.77%
F	9.26%	0.10%	27.37%	0.00%	2.55%	55.77%	6.99%	0.39%	46.88%	0.00%	11.55%	57.14%	0.00%	12.34%	63.74%	0.00%	0.00%	0.00%	0.00%	22.22%	64.41%	0.00%	2.23%	57.47%	78.65%	69.09%	80.00%

* **P** represents *Precision*, **R** represents *Recall*, and **F** represents *F1 score*.

To evaluate SliSE's capability in detecting Reentrancy vulnerabilities within complex contracts, we conducted experiments comparing it with eight state-of-the-art tools using the DB1 dataset. The results, presented in Table 2, demonstrate the detection performance of different tools. It is evident that existing tools achieve a maximum F1 score of less than 10%, and most of them correctly detect 0 Reentrancy vulnerabilities. This underscores the limitations of existing tools in effectively detecting vulnerabilities in real-world, complex DApps. In contrast, SliSE perform the best, achieving an

impressive F1 score of 78.65% on DB1, a significant improvement compared to the highest F1 score of 9.26% achieved by existing tools.

Table 3 compares the detection times of different tools. SliSE takes an average of 25.26 seconds for DB1 detection. While this is higher than static analysis tools like Slither, it falls within a moderate range compared to most symbolic execution tools.

The analysis of experimental results reveals that many existing symbolic execution tools face challenges in precisely searching paths while detecting vulnerabilities in complex contracts. This is due to their limited state exploration capabilities, resulting in incomplete path identification and the omission of numerous paths. This inefficiency slows down program execution and generates a notable number of false negatives. Therefore, precise path pruning is essential for efficient and effective symbolic execution vulnerability detection.

Table 3. Comparison of average detection times

Dataset	Mythril	Securify	Slither	Oyente	Osiris	Manticore	Smartian	Sailfish	Ours
DB1	157.17	26.89	7.11	8.76	14.15	189.8	238.79	7.04	25.26
DB2	276.36	101.18	4.97	13.53	154.54	290.77	298.38	1.19	6.01
DB3	230.49	84	4.37	9.33	141.84	248.63	264.11	5.32	1.94

Answer to RQ1: SliSE outperformed other tools in detecting vulnerabilities within complex contracts, achieving an impressive F1 score of 78.65%, significantly surpassing the maximum of 9.26% achieved by other tools. The precise path pruning techniques employed by SliSE are instrumental in ensuring effective and efficient vulnerability detection.

4.3 Effectiveness of Detecting Reentrancy on Ethereum Contracts

To evaluate SliSE's effectiveness in detecting Reentrancy vulnerabilities on Ethereum contracts, we conducted experiments with DB2 and DB3. These datasets originated from previous research on smart contract security [31, 46], containing 68,610 contracts that were manually annotated and widely recognized as the ground truth for Reentrancy vulnerabilities on Ethereum contracts.

We conducted comparative experiments using eight existing tools, as shown in Table 2. SliSE's superiority is evident, achieving detection recall rates exceeding 90%. On DB2, it significantly outperforms existing tools, demonstrating robust detection capabilities for complex contracts. Notably, apart from Manticore, existing tools exhibit relatively average performance on DB3, with most F1 scores reaching 45% or higher. However, when dealing with the more complex DB2, the vulnerability detection capabilities of existing tools notably decline. The majority of these tools yield F1 scores below 15% due to a high number of false positives. Furthermore, in the time comparison for DB2 and DB3 in Table 3, symbolic execution tools like Mythril and Manticore still struggle with path explosion issues, with average detection times exceeding 230 seconds. In contrast, SliSE's detection time is comparable to that of static analysis tools, averaging less than 7 seconds.

```
function withdrawAll() external {
    uint256 amount = address(this).balance;
    bool success = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
}
```

Fig. 10. False positive of Reentrancy without economic loss

Through analysis, we have identified that false positives (FP) are primarily associated with *specific semantic designs*, which refer to program implementations tailored for specific scenarios.

One example is *Reentrancy can occur without causing economic losses*, meaning that Reentrancy logic exists in the contract but does not result in any losses to the victim. These situations often arise from specific program design choices that prevent Reentrancy but require contextual analysis. SliSE doesn not account for these special semantic designs, leading to false positives. In Figure 10, the code satisfies the conditions for a Reentrancy vulnerability, allowing Reentrancy through the *call.value()* in L3. However, at this point, the contract's balance has already been fully transferred, causing the transaction to revert due to insufficient balance during the Reentrancy. In reality, no economic loss occurs. The lack of specific semantic analysis significantly results in false positives in the detection outcomes.

Answer to RQ2: SliSE's advantage of detecting Reentrancy vulnerabilities in Ethereum smart contracts is evident when compared to existing tools. It consistently achieves a recall rate exceeding 90% and maintains higher F1 scores compared to the best results achieved by existing tools.

4.4 Impact of Pruning in Stage I

In Stage I, SliSE performs program slicing to analyze the contract's I-PDG, pruning irrelevant code segments for targeted analysis. To evaluate the impact of pruning in Stage I on overall detection, we conducted comparative experiments in two modes: *Stage II* and *Stage I & II combined* (representing without and with Stage I). By comparing results with and without pruning in Stage I, we assessed its effect, as shown in Table 4. The impact of pruning is minimal when dealing with datasets of low complexity, such as DB2 and DB3. Pruning in Stage I results in less than a 10% increase in the overall F1 score on DB3. However, for complex contract vulnerability detection on DB1, Stage I pruning significantly enhances accuracy. The F1 score improves from 6.59% to 78.65%, achieving a 11 times increase in results.

Table 4. Statistics of Detection Results by Different Tools

Dataset	DB1			DB2			DB3		
Tools	**Stage I**	**Stage II**	**Stage I & II**	**Stage I**	**Stage II**	**Stage I & II**	**Stage I**	**Stage II**	**Stage I & II**
# TP	70	3	70	38	16	38	30	25	30
# FP	78	7	27	146	11	31	56	1	14
# FN	11	78	11	3	25	3	1	5	1
# TN	736	807	787	21025	21160	21140	56	112	98
Precision	47.30%	30.00%	72.16%	20.65%	59.26%	55.07%	34.88%	96.15%	68.18%
Recall	86.42%	3.70%	86.42%	92.68%	39.02%	92.68%	96.77%	83.33%	96.77%
F1	61.14%	6.59%	78.65%	33.78%	47.06%	69.09%	51.28%	89.29%	80.00%

We also evaluated the impact of pruning in Stage I on the overall detection time, as shown in Figure 11. Pruning significantly reduces the execution time. For complex contract vulnerability detection (DB1), the average detection time decreases from 157.17s to 25.26s, greatly improving the efficiency of detecting vulnerabilities in complex contracts. This reduction in execution time highlights the importance of precise pruning in Stage I for efficient vulnerability detection.

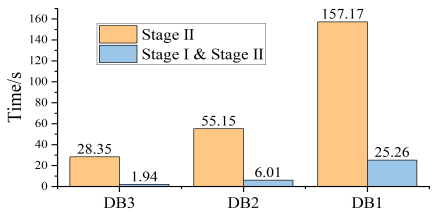

Fig. 11. Impact of pruning

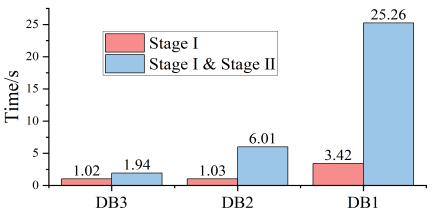

Fig. 12. Impact of symbolic execution verification

Answer to RQ3: Pruning in Stage I significantly improves detection accuracy, boosting the F1 score from 6.59% to 78.65%. Additionally, the precise path pruning in Stage I ensures the efficiency of vulnerability detection.

4.5 Impact of Symbolic Execution Verification in Stage II

In Stage II, SliSE uses symbolic execution to validate the reachability of suspicious vulnerability paths, reducing false positives in the detection results. To evaluate the impact of symbolic execution verification in Stage II for overall detection, we conducted comparative experiments by comparing the results between *Stage I* and *Stage I & II* modes (representing without and with Stage II). Table 4 presents the statistics for Stage I and Stage I & II. The data clearly show that Stage II symbolic execution verification significantly reduces the number of false positives (FPs) in the results. For Reentrancy vulnerability detection on DB1 only with Stage I, there were 78 FPs. When combined with Stage II symbolic execution verification, the number of FPs notably decreased, resulting in a substantial increase in precision from 47.30% to 72.16%. This highlights that the symbolic execution verification in Stage II ensures more precise vulnerability detection results.

We also analyzed the time overhead associated with Stage II's symbolic execution verification. As shown in Figure 12, the overhead introduced by symbolic execution verification is less than 7 seconds on DB2, DB3, with the average detection time remaining at 25.26 seconds for DB1. This time investment is significantly lower compared to the runtime of most symbolic execution tools and is suitable for batch detection purposes.

Answer to RQ4: Symbolic execution verification in Stage II significantly reduces FPs, boosting precision from 47.30% to 72.16% on DB1. Moreover, symbolic execution verification incurs some time overhead, the overall detection time remains reasonable, ensuring efficient detection processes.

5 DISCUSSION

5.1 Capability Analysis of Existing Tools

Zheng et al. [45] conducted an analysis of 1,322 open-source DApps audit reports from 30 security audit teams, compiling a large-scale, fair, real-world smart contract vulnerability dataset known as DB1. Their experiments revealed that existing tools performed poorly in terms of effectiveness and successful detection rates, suggesting that future development should prioritize real-world complex contracts over simple toy contracts. To assess the capability of existing tools to detect complex contracts, we quantified the complexity of these contracts and conducted comparative analysis.

Table 5. Detection capability of existing tools for complex contracts

	AvgCyclomatic		MaxCyclomatic		SumCyclomatic		MaxNesting		CountContractCoupled	
	Detected	Avg.	Detected	Avg.	Detected	Avg.	Detected	Avg.	Detected	Avg.
Mean	1.3(36.1%)	3.6	4.0(18.7%)	21.4	39.5(15.6%)	253.5	1.8(75.0%)	2.4	0.8(34.8%)	2.3
Max	2.0(25.3%)	7.9	11.0(14.9%)	74.0	104.0(11.0%)	949.0	4.0(33.3%)	12.0	3.0(25.0%)	12.0
Median	1.2(36.4%)	3.3	2.0(12.1%)	16.5	25.0(14.6%)	171.0	1.0(50.0%)	2.0	0.0(0.0%)	2.0
Std.	0.3(20.0%)	1.5	3.2(19.8%)	16.2	34.2(14.5%)	236.1	1.5(88.2%)	1.7	1.3(48.1%)	2.7

* *Detected* means successfully detected by existing tool, *Avg.* means the overall average complexity.

We utilized the *Complexity Metrics* proposed by Chao et al. [29] to quantify the complexity of contracts and assess the detection capabilities of existing tools on complex contracts. These metrics provide insights into the complexity and interdependencies of functions and contracts. Specifically, *AvgCyclomatic* calculates the average cyclomatic complexity among functions, *MaxCyclomatic* identifies the most complex function, and *SumCyclomatic* aggregates complexities across functions. *MaxNesting* reveals the deepest nesting of control structures in functions, while *CountContract-Coupled* counts interconnected contracts, indicating their dependencies. We compared the average

complexity metrics of contracts successfully detected by existing tools with the overall average complexity metrics of DB1, as shown in Table 5. The data in the table reveals that only around 30% of the overall contract complexity can be successfully detected by these tools. This highlights the limited capabilities of existing tools in identifying vulnerabilities in real-world complex contracts. They are insufficient for addressing the security needs of practical contracts in DApps, indicating a significant challenge in detecting Reentrancy vulnerabilities in complex contracts.

5.2 Threats to Validity

External Validity. We validated SliSE's Reentrancy vulnerability detection performance on Ethereum using the DB2 and DB3 datasets. DB2 collected from positive labels in existing tool detection results on Ethereum contracts, without considering negative labels. However, labeling all Ethereum contracts is time-consuming and error-prone, which demands a substantial team of experienced engineers for Reentrancy vulnerability analysis. DB2 was created by subjecting 230,548 verified smart contracts from Etherscan to scans by five automated detection tools [1, 8, 9, 25, 37], followed by two rounds of manual examination, Zheng et al. underscored the error-prone nature of vulnerability labeling [46]. In addition, DB3 is the well-known SmartBugs dataset [31]. These datasets are widely acknowledged and have been openly shared at top software engineering conferences, establishing them as reliable sources of ground truth for Ethereum contracts.
Internal Validity. While various permission controls in smart contracts are essential for security, they present a challenge that demands specific semantic comprehension to assess path feasibility. SliSE does not account for these specific semantic requirements, resulting in false positives in detection results. To expand the scope of vulnerability detection, we draw inspiration from a range of patterns in *path protective techniques* (PPTs) [40]. These patterns have been identified through empirical research and help identify common false positives in state-of-the-art tool rules. By applying these patterns, we effectively reduce false positives, thereby improving scalability and effectiveness.

6 RELATED WORK

6.1 Reentrancy Vulnerability Detection

There are numerous tools that support Reentrancy vulnerability detection, utilizing various vulnerability detection techniques, including static analysis, symbolic execution, fuzz testing, and formal verification. Tools like Oyente [25], Osiris [36] utilize CFG-based bytecode analysis with pattern matching for fast and scalable Reentrancy vulnerability detection. Slither [12] generates intermediate language *SlithIR* by analyzing smart contract AST, employing rule-based detection for vulnerabilities, and scaling up to 5 types of Reentrancy vulnerabilities. Clairvoyance [40] introduces taint analysis via shadow stacks to reduce false positives while combining lightweight symbolic execution for vulnerability detection. Mythril [9], Manticore [27] use emulation testing to generate sequences of transactions triggering vulnerabilities. MPro [42] simulates execution by increasing the depth of symbolic execution analysis, exploring deeper levels of the state space. Smartian [8] applies data flow analysis to guide path exploration, generating critical transaction sequences triggering vulnerabilities. Sailfish [1] analyzes program dependency information to generate *Storage Dependency Graph* (SDG) detects dangerous access patterns, but can not analyze global cross-contract data flow. Pluto [26] supports cross-contract vulnerability detection by constructing *Inter-Contract Control Flow Graph* (I-CFG) to enable cross-contract bytecode traversal, then exploring I-CFG to detect vulnerabilities. Park [43] proposed a method based on parallel symbolic execution, which proposed a dynamic forking algorithm based on process forking to speed up vulnerability detection.

Differing from existing methods. SliSE constructs the *Inter-contract Program Dependency Graph* (I-PDG) to provide global control and data dependencies within contracts, including cross-contract data flows. In contrast, Sailfish [1] exclusively conducts program dependency analysis on state variables within single contract and does not support cross-contract analysis. EtherSolve [10] calculates dynamic jump address using the *symbolic stack*, which always fails in complex contracts due to issues with excessive recursion depth. SliSE leverages constant propagation analysis with SSA to quickly compute dynamic jump addresses and recovery the CFG paths. Compared to Pluto [26], SliSE primarily slices out code blocks related to Reentrancy vulnerabilities from complex contracts for focused analysis, employing symbolic execution for efficient and reliable verification. Evaluation results show that SliSE outperforms many state-of-the-art tools in detecting Reentrancy vulnerabilities in complex contracts.

7 CONCLUSION

In this paper, we introduce a tool named SliSE, designed for efficient detection of Reentrancy vulnerabilities in complex contracts. The detection process consists of two stages: Warning Search and Symbolic Execution Verification. In the Warning Search stage, SliSE analyzes the *Inter-contract Program Dependency Graph* (I-PDG) with program slicing, collecting suspicious vulnerability information as warnings. In the Symbolic Execution Verification stage, it employs symbolic execution to traverse the paths indicated by the warning information and validate their reachability, ensuring effective vulnerability detection. In comparative experiments, SliSE achieves impressive results, with an F1 score of 78.65%, surpassing the highest score of 9.26% achieved by eight existing state-of-the-art tools. Additionally, it achieves a recall rate exceeding 90% for Reentrancy vulnerability detection of contracts on Ethereum. Overall, SliSE provides an effective solution for detecting reentrancy vulnerabilities in complex contracts.

ACKNOWLEDGMENTS

The work described in this paper is supported by the National Natural Science Foundation of China (62032025, 62302534, 62332004), and the Major Key Project of Peng Cheng Laboratory under Grant PCL2023A05-2.

REFERENCES

[1] 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. https://doi.org/10.1109/SP46214.2022.9833721

[2] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. 1998. Conditioned program slicing. *Information and Software Technology* 40, 11-12 (1998), 595–607.

[3] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C. Myers. 2021. Compositional Security for Reentrant Applications. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1249–1267. https://doi.org/10.1109/SP40001.2021.00084

[4] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Ben Livshits. 2023. Smart contract and defi security: Insights from tool evaluations and practitioner surveys. *arXiv preprint arXiv:2304.02981* (2023).

[5] Jiachi Chen, Mingyuan Huang, Zewei Lin, Peilin Zheng, and Zibin Zheng. 2023. To Healthier Ethereum: A Comprehensive and Iterative Smart Contract Weakness Enumeration. arXiv:cs.SE/2308.10227

[6] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2022), 327–345. https://doi.org/10.1109/TSE.2020.2989002

[7] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1503–1520. https://doi.org/10.1145/3319535.3345664

[8] 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. https://doi.org/10.1109/ASE51524.2021.9678888

[9] ConsenSys. 2020. Mythril. https://github.com/ConsenSys/mythril

[10] Filippo Contro, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. 2021. Ethersolve: Computing an accurate control-flow graph from ethereum bytecode. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 127–137.

[11] Phil Daian. 2016. Analysis of the DAO exploit. https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[12] 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)*. IEEE, 8–15.

[13] 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*. 728–739.

[14] Seungwon Go. 2018. Smart Contract : Security Patterns. https://medium.com/returnvalues/smart-contract-security-patterns-79e03b5a1659

[15] Daojun Han, Qiuyue Li, Lei Zhang, Tao Xu, et al. 2023. A Smart Contract Vulnerability Detection Model Based on Syntactic and Semantic Fusion Learning. *Wireless Communications and Mobile Computing* 2023 (2023).

[16] Mary Jean Harrold, Brian Malloy, and Gregg Rothermel. 1993. Efficient construction of program dependence graphs. *ACM SIGSOFT Software Engineering Notes* 18, 3 (1993), 160–170.

[17] 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 (CCS '19)*. Association for Computing Machinery, New York, NY, USA, 531–548. https://doi.org/10.1145/3319535.3363230

[18] Sebastian Holler, Sebastian Biewer, and Clara Schneidewind. 2023. HoRStify: Sound Security Analysis of Smart Contracts. *arXiv preprint arXiv:2301.13769* (2023).

[19] insurgent. 2022. Solidity Smart Contract Security: 4 Ways to Prevent Reentrancy Attacks. https://betterprogramming.pub/solidity-smart-contract-security-preventing-reentrancy-attacks-fc729339a3ff

[20] Thomas Shababi Jacques Dafflon, Jordi Baylina. 2017. ERC-777: Token Standard. https://eips.ethereum.org/EIPS/eip-777

[21] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/3238147.3238177

[22] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.

[23] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. 1317–1333.

[24] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. 2022. SmartDagger: A Bytecode-Based Static Analysis Approach for Detecting Cross-Contract Vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 752–764. https://doi.org/10.1145/3533767.3534222

[25] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[26] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. 2021. Pluto: Exposing vulnerabilities in inter-contract scenarios. *IEEE Transactions on Software Engineering* 48, 11 (2021), 4380–4396.

[27] 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 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189.

[28] 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 (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 778–788. https://doi.org/10.1145/3377811.3380334

[29] Chao Ni, Cong Tian, Kaiwen Yang, David Lo, Jiachi Chen, and Xiaohu Yang. 2023. Automatic Identification of Crash-inducing Smart Contracts. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 108–119. https://doi.org/10.1109/SANER56733.2023.00020

[30] Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan Karame, and Lucas Davi. 2023. EF/CF: High Performance Smart Contract Fuzzing for Exploit Generation. *arXiv preprint arXiv:2304.06341* (2023).

[31] smartbugs. 2020. Smartbugs wild dataset. https://github.com/smartbugs/smartbugs-wild

[32] 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)*. 1361–1378.

[33] 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 *37th IEEE/ACM International Conference on Automated Software Engineering (ASE22)*. Association for Computing Machinery, New York, NY, USA, Article 36, 12 pages. https://doi.org/10.1145/3551349.3560429

[34] Phuwanai Thummavet. 2022. Solidity Security By Example 04: Cross-Function Reentrancy. https://medium.com/valixconsulting/solidity-smart-contract-security-by-example-04-cross-function-reentrancy-de9cbce0558e

[35] Phuwanai Thummavet. 2022. Solidity Security By Example 05: Cross-Contract Reentrancy. https://medium.com/valixconsulting/solidity-smart-contract-security-by-example-05-cross-contract-reentrancy-30f29e2a01b9

[36] 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.

[37] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

[38] Philippe Castonguay Witek Radomski, Andrew Cooke. 2018. ERC-1155: Multi Token Standard. https://eips.ethereum.org/EIPS/eip-1155

[39] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[40] Jiaming Ye, Mingliang Ma, Yun Lin, Yulei Sui, and Yinxing Xue. 2020. Clairvoyance: Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 274–275.

[41] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. 2020. TXSPECTOR: Uncovering Attacks in Ethereum from Transactions. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 156, 18 pages.

[42] 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, 456–462.

[43] 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 (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 740–751. https://doi.org/10.1145/3533767.3534395

[44] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2024. A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends. arXiv:cs.SE/2311.10372

[45] Zibin Zheng, Jianzhong Su, Jiachi Chen, David Lo, Zhijie Zhong, and Mingxi Ye. 2023. DAppSCAN: Building Large-Scale Datasets for Smart Contract Weaknesses in DApp Projects. arXiv:cs.SE/2305.08456

[46] Zibin Zheng, Neng Zhang, Jianzhong Su, Zhijie Zhong, Mingxi Ye, and Jiachi Chen. 2023. Turn the Rudder: A Beacon of Reentrancy Detection for Smart Contracts on Ethereum. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, 295–306. https://doi.org/10.1109/ICSE48619.2023.00036

[47] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2444–2461.

[48] Shunfan Zhou, Zhemin Yang, Jie Xiang, Yinzhi Cao, Min Yang, and Yuan Zhang. 2020. An Ever-Evolving Game: Evaluation of Real-World Attacks and Defenses in Ethereum Ecosystem. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 157, 17 pages.

Received 2023-09-27; accepted 2024-01-23