VULSEYE: Detect Smart Contract Vulnerabilities via Stateful Directed Graybox Fuzzing

Ruichao Liang, Jing Chen, *Senior Member, IEEE*, Cong Wu, Kun He, *Member, IEEE*, Yueming Wu, Ruochen Cao, Ruiying Du, Ziming Zhao, and Yang Liu, *Senior Member, IEEE*

***Abstract*—Smart contracts, the cornerstone of decentralized applications, have become increasingly prominent in revolutionizing the digital landscape. However, vulnerabilities in smart contracts pose great risks to user assets and undermine overall trust in decentralized systems. Fuzzing, a prominent security testing technique, is extensively explored to detect vulnerabilities. But current smart contract fuzzers fall short of expectations in testing efficiency for two primary reasons. Firstly, smart contracts are stateful programs, and existing approaches, primarily coverage-guided, lack effective feedback from the contract state. Consequently, they struggle to effectively explore the contract state space. Secondly, coverage-guided fuzzers, aiming for comprehensive program coverage, may lead to a wastage of testing resources on benign code areas. This wastage worsens in smart contract testing, as the mix of code and state spaces further complicates comprehensive testing. To address these challenges, we propose VULSEYE, a stateful directed graybox fuzzer for smart contracts guided by vulnerabilities. Different from prior works, VULSEYE achieves stateful directed fuzzing by prioritizing testing resources to code areas and contract states that are more prone to vulnerabilities. We introduce *Code Targets* and *State Targets* into fuzzing loops as the testing targets of VULSEYE. We use static analysis and pattern matching to pinpoint *Code Targets*, and propose a scalable backward analysis algorithm to specify *State Targets*. We design a novel fitness metric that leverages feedback from both the contract code space and state space, directing fuzzing toward these targets. With the guidance of code and state targets, VULSEYE alleviates the wastage of testing resources on benign code areas and achieves effective stateful fuzzing. In comparison with state-of-the-art fuzzers, VULSEYE demonstrated superior effectiveness and efficiency. Notably, it uncovered 4,845 vulnerabilities in 42,738 real-world smart contracts, outperforming existing approaches by up to 9.7×, and identified 11 previously unknown vulnerabilities within the top 50 Ethereum DApps, involving approximately 2,500,000 USD.**

***Index Terms*—Fuzz testing, static analysis, smart contract.**

Received 9 July 2024; revised 12 November 2024 and 11 December 2024; accepted 20 January 2025. Date of publication 3 February 2025; date of current version 24 February 2025. This work was supported in part by the National Key Research and Development Program of China under Grant 2021YFB2700200; in part by the National Natural Science Foundation of China under Grant 62172303 and Grant 62472323; in part by the Key Research and Development Program of Hubei Province under Grant 2022BAA039 and Grant 2024BAB018; in part by Wuhan Scientific and Technical Achievements Project under Grant 2024030803010172; in part by the Key Research and Development Program of Shandong Province under Grant 2022CXPT055; in part by the Rizhao Natural Science Foundation under Grant RZ2021ZR4; in part by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity Research and Development Programme under Grant NCRP25-P04-TAICeN; in part by DSO National Laboratories under the AI Singapore Programme under Award AISG2-GC-2023-008; in part by the Cyber Security Agency through the National Cybersecurity Research and Development Programme under Grant NCRP25-P04-TAICeN; and in part by the National Research Foundation, Prime Minister's Office, Singapore, under the Campus for Research Excellence and Technological Enterprise (CREATE) programme. The associate editor coordinating the review of this article and approving it for publication was Dr. Alessandro Brighente. *(Corresponding author: Cong Wu.)*

Ruichao Liang, Jing Chen, Cong Wu, Kun He, Ruochen Cao, and Ruiying Du are with the Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China (e-mail: liangruichao@whu.edu.cn; chenjing@whu.edu.cn; cnacwu@whu.edu.cn; hekun@whu.edu.cn; crc2019@whu.edu.cn; duraying@whu.edu.cn).

Yueming Wu is with the Cyber Security Laboratory, College of Computing and Data Science, Nanyang Technological University, Singapore 639798 (e-mail: wuyueming21@gmail.com).

Ziming Zhao is with the Khoury College of Computer Sciences, Northeastern University, Boston, MA 02115 USA (e-mail: z.zhao@northeastern.edu).

Yang Liu is with the Cyber Security Laboratory, College of Computing and Data Science, Nanyang Technological University, Singapore 639798, and also with China-Singapore International Joint Research Institute (CSIJRI), Guangzhou 510000, China (e-mail: yangliu@ntu.edu.sg).

Digital Object Identifier 10.1109/TIFS.2025.3537827

I. INTRODUCTION

SMART contracts, self-executing programs that automate the requisite actions of agreements or contracts, form a cornerstone of the blockchain and cryptocurrency ecosystem. However, vulnerabilities within these contracts pose considerable risks [1], exemplified by the DAO incident, where a reentrancy vulnerability led to a loss of $150 million [2], and the Poly Network breach, resulting in a $611 million loss [3]. Thus, it is imperative to conduct thorough security testing on smart contracts to detect vulnerabilities and prevent significant financial and operational consequences.

Fuzz testing, an automated technique for program testing, identifies vulnerabilities by generating random inputs and systematically exploring a program's space [4]. Various efforts have been undertaken to apply fuzz testing techniques to smart contracts [5], [6], [7], [8], [9], [10], [11], [12]. These initiatives involve generating inputs that mimic real-world transactions for testing smart contracts, with a focus on coverage-guided greybox fuzzing to improve code coverage and facilitate vulnerability discovery. For example, sFuzz [8] introduces an adaptive strategy for seed selection aimed at reaching hard-to-cover branches for higher overall code coverage. Smartian [10] utilizes data-flow-based feedback for generating high-quality test cases in order to enhance the code coverage. Despite these efforts, there still remain following significant gaps.

A. Insufficient State Space Exploration in Smart Contract Fuzzing

Although existing contract fuzzers have thoroughly studied the use of feedback from code space, such as code coverage, to guide fuzzing, they lack an effective feedback mechanism for contract states, resulting in indiscriminate exploration of the contract state space. As smart contracts are intrinsically

stateful [13], with state variables playing a pivotal role in their functionality, accurately identifying and examining specific states is essential for vulnerability exploitation. Existing approaches such as ConFuzzius [9] and Harvey [7] typically generate contract states through combinations of transactions. ITYFuzz [14] utilizes snapshot for contract state to reduce the time required for transaction re-execution. However, they are insufficient for a targeted and systematic examination of the contract state space, thereby impeding effective vulnerability detection.

B. Inefficient Code Space Exploration in Smart Contract Fuzzing

Most existing contract fuzzers adopt the coverage-guided fuzzing strategy, aiming for comprehensive program coverage. However, research has shown that vulnerabilities within programs tend to be concentrated in specific code areas, leaving the majority of code benign [15]. Therefore, coverage-guided fuzzers may waste testing resources on these non-vulnerable areas [16]. This issue becomes more pronounced in smart contract testing, where the complex interaction between contract state and code spaces complicates comprehensive testing, resulting in a significant portion of testing efforts being allocated to benign code areas and contract states.

To bridge the gaps, we aim to design a stateful directed fuzzer for smart contracts, inspired by the concept of directed greybox fuzzing (DGF) [16], [17]. DGF, known for its efficiency in allocating testing resources to specific areas of interest while minimizing unnecessary stress on unrelated parts, has shown effectiveness in various contexts [17], [18], [19], [20]. However, existing DGFs lack capabilities for stateful fuzzing, primarily due to their focus on the code space while overlooking the essential state space in smart contracts. Moreover, these tools depend heavily on external information like bug reports for target identification [16], limiting their autonomy in identifying smart contract vulnerabilities. Different from the existing DGFs, our motivations include: i) achieving stateful fuzzing in the contract state space through the use of feedback from critical states, and ii) enabling directed fuzzing in the contract code space via automated target identification.

Our Solution: In this paper, we propose VULSEYE, a directed graybox fuzzer for smart contracts with state awareness, which we refer to as a stateful directed fuzzer. It prioritizes resources on vulnerable code areas and contract states, enhancing the efficiency of vulnerability detection. We utilize static analysis and pattern matching to identify code areas vulnerable to exploits, termed as *code targets*. We design a backward analysis algorithm for bytecode to determine the critical contract states required for vulnerability exploitation, termed as *state targets*. Code and state targets provide feedback in both code and state levels during the fuzzing process of VULSEYE. We propose a novel fitness metric integrating the feedback from code and state levels to achieve directed fuzzing across both contract code and state spaces.

We evaluated VULSEYE's performance on both closed datasets and real-world scenarios. The evaluation reveals that VULSEYE outperforms SOTA tools in vulnerability detection and proves effective in identifying real-world vulnerabilities. Notably, VULSEYE detected 4,845 vulnerabilities among 42,738 real-world smart contracts, outperforming SOTA tools by up to 9.7×. Besides, utilizing VULSEYE, we found 11 previously unknown vulnerabilities in the top 50 Ethereum DApps and have reported them to corresponding authorities.

Contributions: This paper makes following contributions.

- We propose VULSEYE, the first stateful directed graybox fuzzer for smart contracts. It autonomously identifies vulnerable code areas and contract states, then prioritizes testing resources on these targets, achieving stateful and directed exploration of both contract code and state spaces.
- We design a backward analysis algorithm for contract bytecode. It efficiently identifies vulnerable states to specify state targets and provides feedback on the contract state space throughout the fuzzing process.
- We contribute a fitness metric that evaluates the proximity of seeds to testing targets at the contract code and state levels, effectively gauging the potential of seeds to trigger vulnerabilities. This algorithm guides seed scheduling in fuzzing, facilitating targeted vulnerability detection within both code and state spaces.
- We comprehensively evaluated VULSEYE and benchmarked it against SOTA tools. The results show that: i) VULSEYE reaches specific test targets up to 8.4× faster than the baselines. ii) It outperforms SOTA tools in terms of both efficiency and quantity on a ground truth vulnerability dataset. iii) It identified 4,845 vulnerabilities among 42,738 real-world smart contracts, surpassing SOTA tools in both quantity (9.7×) and true positive rate. iv) Additionally, it uncovered 11 previously unknown vulnerabilities in the top 50 Ethereum DApps, involving about 2,500,000 USD.

The implementation of VULSEYE and our datasets are available at: https://github.com/SCFuzzing/Vulseye.

II. BACKGROUND

A. Stateful Smart Contract

Smart contracts [22] are programs running on top of blockchain platforms. They are Turing-complete programs used to automate the execution of an agreement and process assets on blockchain [23]. Within smart contracts, there are a kind of variables called state variables that are created at the contract level and stored permanently on the blockchain [24]. The values of state variables can change based on the contract's execution, provide a direct reflection of the contract's states, and impact the contract's behavior and functionality. State variables make smart contracts *stateful* programs by preserving and managing their states throughout their lifecycle. To execute smart contracts, their high-level code is compiled into bytecode, a machine-readable format that runs on virtual machines like the Ethereum Virtual Machine (EVM). Bytecode encodes all instructions for contract operations, including state updates, function execution,

TABLE I
COMPARISON OF REPRESENTATIVE RELATED WORKS

Features	Data Dependency†	Directed Seed Mutation	Adaptive Seed Priority	Exploring Code Space	Exploring State Space‡
CONTRACTFUZZER [21]	○	○	○	◐	○
HARVEY [7]	○	●	○	●	◐
SFUZZ [8]	○	◐	○	●	○
CONFUZZIUS [9]	●	○	◐	●	◐
SMARTIAN [10]	●	○	○	●	○
IR-FUZZ [12]	●	○	●	●	○
ITYFUZZ [14]	○	○	●	●	◐
VULSEYE	●	●	●	●	●

* ●= true, ◐= partially true, ○= false.
† Support data dependency analysis to generate meaningful transaction sequences.
‡ Support testing on specific contract states.

and interactions with other contracts or blockchain components. Each instruction in the bytecode is directly associated with a specific opcode, which the virtual machine interprets and executes step by step.

B. Graybox Fuzzing

Greybox fuzzing is a software testing method that combines aspects of white-box [25] and black-box testing [26]. It involves injecting semi-random input data, known as seeds, into a program to uncover vulnerabilities and potential issues [27], [28]. Based on the feedback information from the execution of the program under test cases, greybox fuzzers generate new inputs and perform seed scheduling to manage and prioritize these inputs to effectively test the program. Most greybox fuzzing tools are coverage-guided, striving to explore as many program paths as possible [16].

C. Limitations of Existing Contract Fuzzers

Firstly, being stateful programs, smart contracts differ from typical programs by not only having their code space but also having an implicit state space. This complexity adds a layer of difficulty to the exploration of the program space to detect vulnerabilities [29]. Some existing fuzzers take into account the stateful nature of smart contracts. For instance, some [7], [9], [10], and [12] attempt to manipulate the contract state by leveraging the read-write relationship of variables to create transaction sequences, while some [14] employ snapshot to preserve the contract state for rapid resumption. However, these approaches lack knowledge of the specific state required to trigger vulnerabilities, resulting in unguided and inefficient exploration of the contract state space, as shown in Table I. Secondly, existing tools predominantly rely on coverage-guided fuzzing strategies to achieve greater code coverage. However, this may result in an inefficient allocation of effort on testing non-vulnerable code areas, while overlooking critical code areas. The stateful nature of smart contracts further compounds this issue. These limitations are exemplified in Section III. As shown in Table I, VULSEYE surpasses existing state-of-the-art (SOTA) fuzzers by facilitating targeted exploration in both the code and state spaces of smart contracts. These advancements collectively lead to more efficient vulnerability detection.

```
1  contract FancyBank{
2    mapping(address => uint256) private balances;
3    uint256 dueDate = 0, unlock = 0;
4    event WithdrawalFailed(address user, uint256 amount);
5
6    function deposit(uint256 amount) public payable{
7      require(msg.value >= amount)
8      balances[msg.sender] += amount;}
9
10   function setState(uint256 time,uint256 State) public{
11     dueDate = time;
12     unlock = State;}
13
14   function withdraw(uint256 amount) public {
15     require(balance[msg.sender] >= amount);
16     if(dueDate >30 && dueDate < 40 && unlock == 1){
17       msg.sender.call{value: amount}();
18       balance[msg.sender] -= amount;
19     }else{
20       emit WithdrawalFailed(msg.sender, amount);}}
21   /** Other functions */}
```

Fig. 1. Motivating Example.

III. MOTIVATING EXAMPLE

This section presents a motivating example: a contract that allows users to deposit funds and withdraw them as needed. The contract's simplified segment is illustrated in Figure 1. It is vulnerable to reentrancy because of the `withdraw()` function. Intended for users to withdraw funds, this function mistakenly transfers funds before updating the `balance` variable. This allows an attacker to reenter the `withdraw()` function multiple times during the external call in Line 17, ultimately draining the contract's funds. However, it is not easy to exploit this vulnerability in practice, as illustrated below.

A. State Feedback Is Necessary

As discussed in Section I, stateful programs typically necessitate reaching specific states before the vulnerability is manifested. In the contract depicted in Figure 1, invoking the `setState()` function to set the contract states to: `dueDate`$\in$`(30,40)` and `unlock`$\in${1} is necessary before triggering the reentrancy vulnerability in Line 17.

Now considering a scenario with three seeds, each seed comprises two transactions triggering two functions:

Seed A: ①`deposit(10)` ⟶ ②`withdraw(10)`
Seed B: ①`setState(100,0)` ⟶ ②`withdraw(10)`
Seed C: ①`setState(50,1)` ⟶ ②`withdraw(10)`

While all three seeds ultimately follow the *else* branch (Line 19) and miss the vulnerable code location (Line 17), seed C is closer to exploiting the vulnerability. This is because seed C modifies the critical state variables to be closer to the states that are necessary for triggering the vulnerability (i.e., `dueDate`$\in$`(30,40)`, `unlock`$\in${1}). Consequently, allocating more testing resources to seed C during the fuzzing process can expedite the discovery of the vulnerability. However, current approaches such as ConFuzzius [9] and Harvey [7] rely solely on combining transactions in a Read-After-Write order (a trait present in all three seeds in this example) to manipulate the contract state, but they lack feedback from the contract state for systematic stateful exploration. As a result, they fail to recognize `dueDate` and `unlock` as key state variables for triggering the vulnerability, thus hindering their ability to differentiate between the relative effectiveness of these

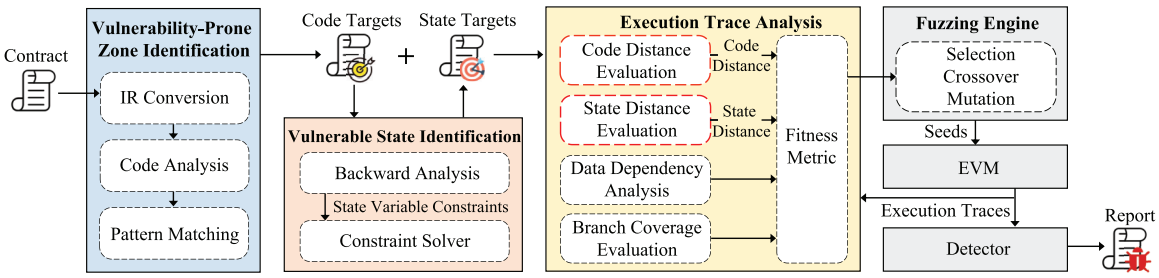


Fig. 2. Overview of VULSEYE.

three seeds. This observation underscores the significance of using feedback from the state space in the fuzzing of stateful programs like smart contracts.

B. Pursuing Comprehensive Coverage May Reduce Efficiency

The vulnerability can be exploited when an attacker manages to pass the *if* condition in Line 16, and execute the external call in Line 17. However, exploiting this vulnerability in the *if* branch is time-consuming due to the strict state conditions it requires. Test cases are more likely to take the *else* branch. In such situations, the current approaches with a coverage-guided fuzzing strategy would typically shift focus to other non-critical functions to enhance overall code coverage, since intensifying code coverage through testing the `withdraw()` function has presented significant challenges. This observation underscores the inefficiency in indiscriminately chasing code coverage, as it often results in disproportionate testing resources being spent on non-vulnerable code sections. A more effective strategy for uncovering vulnerabilities would be to strategically focus testing efforts on code segments with a higher susceptibility to vulnerabilities.

IV. OVERVIEW

We brief threat model and give an overview of VULSEYE.

A. Threat Model

Smart contracts, with their nature of blockchain-based execution and capability of handling digital assets, introduce a set of unique vulnerabilities distinct from traditional platforms. These vulnerability threats often arise from developer oversights or inherent blockchain characteristics, and exploiting them can cause significant and irreversible financial damage. They can be categorized into different types including *Ether Leaking*, *Reentrancy*, *Controlled Delegatecall*, *Suicidal*, etc., which have been well illustrated in previous studies [11], [21]. The attacker has the ability to access data on the public blockchain, analyze smart contracts to identify vulnerabilities, and knows how to exploit them by sending transactions or deploying new contracts.

VULSEYE aims to provide an efficient stateful directed fuzzing solution for vulnerability detection, ensuring the safety of smart contracts before deployment and enabling defenders to timely implement preventive measures for on-chain vulnerable contracts. This will mitigate the threats that contract vulnerabilities pose to blockchain and decentralized ecosystems.

B. Overview of VULSEYE

To implement stateful directed fuzzing in both contract code and state spaces to hunt vulnerabilities, we address three main challenges: ***locating vulnerability-prone code*** in smart contracts, ***identifying critical contract states*** required to exploit vulnerabilities in these code areas, and ***guiding fuzzing to test specific states and code*** for accelerated vulnerability detection.

As shown in Figure 2, to locate vulnerable code areas, a smart contract is first compiled and converted into an Intermediate Representation (IR), which serves as the foundation for subsequent static code analysis [30]. Based on the result of code analysis, VULSEYE conducts pattern matching to identify code areas that could potentially harbor vulnerabilities, referred to as code targets. To identify vulnerable contract states necessary for exploiting vulnerabilities, VULSEYE conducts backward analysis on the control flow graph (CFG) starting from code targets, gathering the constraints associated with state variables. Then, a constraint solver is applied to resolve the state constraints and specify the state targets. During the fuzzing loop, VULSEYE analyzes execution traces, evaluating both the code and state distances of seeds to estimate their propensity to trigger vulnerabilities. Seeds with a greater potential to trigger vulnerabilities are given higher priority in the seed scheduling process, and values from state targets are incorporated into the mutation pool to generate vulnerable contract states. Vulnerability detection is carried out by a detector that evaluates the execution trace based on test oracles.

V. METHODOLOGY

In this section, we detail the design of VULSEYE.

A. Target Identification

To implement directed gray box fuzzing, the first step is to specify potential vulnerable locations as test targets, which, in our solution, involves identifying code and state targets.

1) Vulnerability-Prone Zone Identification: We statically analyze the contracts to identify potentially vulnerable code areas prior to conducting fuzzing, as illustrated in Algorithm 1. We utilize SlithIR [31], an intermediate representation, to represent Solidity code of the contracts under test. A series of code analyses are carried out based on the intermediate representation, such as function property analysis and data dependency analysis (Line 2). Utilizing insights gained from

Algorithm 1 Code Target Identification

```
   Input  : Contract C
   Output: CodeTargets
1: irCode, cfg ← IRConversion (C)
2: res ← CodeAnalysis(irCode, cfg)
3: for node in cfg.nodes do
4:     for ir in node.irs do
5:         offset ← PatternMatching (ir,cfg,res)
6:         CodeTargets ← Locating (offset,cfg)
7:     end
8: end
```

TABLE II
HAZARDOUS BEHAVIORS THAT MAY LEAD TO VULNERABILITIES

Hazardous Behaviors Description	Potential Bugs
Has *Payable* function. No *Suicide* or *Selfdestruct* operation. No high-level or low-level *calls* that send ether.	Lock Ether
Delegatecall using *msg.data* as destination.	Controlled Delegatecall
Delegatecall using *msg.data* as arguments.	Dangerous Delegatecall
Use *block data* in *Require* or *If* statement. Send ether following this statement.	Block Dependency
Read a state variable. External *call* after the *read*. *Write* the same state variable after the *call*.	Reentrancy
No use of *msg.sender* as index. Send ether with no dependency on *msg.value*. Use *msg.sender* or *msg.data* as receiver.	Ether Leaking
Has function that is not protected. Set this function as *public* or *external* visible. Has *Suicide* or *Seftdestruct* in this function.	Suicidal

these analyses, we navigate through the contract's CFG, scrutinizing semantic contract behaviors through pattern matching to identify hazardous behaviors that could potentially lead to vulnerabilities (Lines 3-5). Our pattern matching involves the following key steps. i) The pattern matching process begins with the definition of pattern rules that are designed based on known vulnerability characteristics. We translate them into a set of signature actions or conditions as outlined in Table II. These rules are formulated to capture sequences of operations, control flows, and dependencies indicative of security risks. For instance, a sequence of basic blocks where a state variable is ① initially read before an external call, and ② subsequently written after the external call, may suggest a potential *Reentrancy* vulnerability. ii) To ensure comprehensive detection, we consider different levels of granularity in the code structure. Patterns are applied both at a high level (e.g., function-level behaviors) and within finer structures (e.g., statement or parameters) to maximize coverage. iii) Once a pattern is matched, we pinpoint the specific basic blocks in the contract's CFG as the *code targets* (Line 6). To enhance the detection of vulnerable code areas and reduce the risk of false negatives, we traverse the contract code space and conduct pattern matching with an over-approximation principle. Specifically, patterns are defined with broad criteria to avoid prematurely excluding potentially hazardous areas from analysis, and we disregard reachability constraints when traversing the CFG, allowing thorough inspection across all possible control flows.

It is important to note that, the *code targets* identified in this phase do not directly report vulnerabilities; rather, they highlight potential vulnerable code areas (which may not necessarily result in vulnerabilities). These flagged areas undergo further validation through subsequent fuzzing tests, thereby mitigating the risk of high false positive rates inherent in static analysis [32].

2) Vulnerable State Identification: Smart contracts are stateful programs, and vulnerabilities necessitate specific states for exploitation. While we've identified vulnerable code targets and plan to utilize them as guidance for fuzzing, indiscriminately altering the contract's state can still be inefficient in reaching those code targets and exploiting vulnerabilities. To address this challenge, we introduce a backward analysis algorithm designed to identify vulnerable contract states, facilitating the stateful exploration in the contract state space. Our vulnerable state identification is a fine-grained analysis, which focuse on specific storage slots in the EVM. Since contract bytecode operates at the stack/storage/memory level of granularity, which source code analysis cannot directly access, our backward analysis is conducted at the bytecode level, similar to symbolic execution. But unlike symbolic execution [33], [34], [35] which is slow and suffers from poor scalability due to the path explosion problem, we focus solely on the code targets-related paths and states, ensuring the efficiency and scalability of our tool. Algorithm 2 reveals how we perform backward analysis on the CFG and define the corresponding state targets. Algorithm 3 further details the process by which backward analysis gathers all state constraints for a given path.

Algorithm 2 State Target Identification

```
   Input  : CFG C, Target T
   Output: stateTarget
1: stateCons ← ∅
2: paths ← FindPath (C, T)
3: for p in paths do
4:     pathCons ← 𝕌
5:     branchPoints ← p.branchPoints
6:     for bp in branchPoints do
7:         branchCons ← BackwardAnalysis (p, bp)
8:         pathCons ← pathCons∩ branchCons
9:     end
10:    stateCons ← stateCons∪ pathCons
11: end
12: stateTarget ← Solve (stateCons)
```

Algorithm 3 Contract Bytecode Backward Analysis

```
   Input  : path P, branchPoint B
   Output: branchCons
1: branchCons ← InitCons (P, B)
2: t ← InitTraceBack (P, B)
3: stack ← []
4: reversedIns ← P.BackwardFrom(B)
5: while NotEmpty (t) && NotEmpty (reversedIns) do
6:     ins, pc ← reversedIns.Pop()
7:     stack[pc] ← StackReconstruction (ins, pc, stack)
8:     cons, t ← ConsCollection (ins, pc, stack, t)
9:     branchCons ← branchCons∩ cons
10: end
```

Algorithm 2 takes as input the contract's CFG and the code target obtained from the vulnerability-prone zone identification process, and outputs a set of state targets. We handle loops in the CFG using loop unrolling with a upper bound of 20. This compromise balances efficiency and soundness in our analysis, as smart contracts typically avoid extensive loops due to gas limits. We initialize the set of state constraints

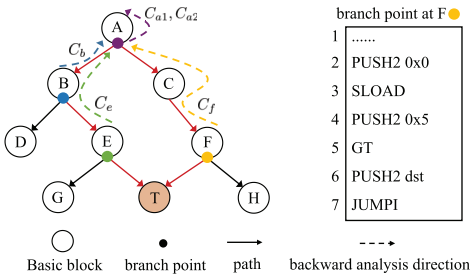


Fig. 3. Example for State Identification.

(`stateCons`) and use NetworkX [36], an efficient tool for manipulating graph structures, to identify all paths leading to the target basic blocks (Lines 1-2). For each path, we extract the involved branch jumps (typically the `JUMPI` instruction) in Line 5 and conduct backward analysis at these branch points, gathering constraints on state variables in Line 7. Next, we intersect these constraints in Line 8 to derive state requirements for this path to reach the code target. The union of these path-specific constraints (Line 10) defines the overall state requirements for all paths. Finally, we employ a constraint solver to resolve these constraints, determining the specific contract states required to reach the code target, referred to as the *state targets*. It is worth noting that for state variables of mapping type that take a symbolic address as the key, we do not regard them as state targets because the key cannot be concretely determined in static analysis.

Algorithm 3 further details the implementation of backward analysis at each branch point. In Line 1, we generate an initial constraint based on the jump direction at the current branch point. To be specific, the jump direction is determined by the second parameter of the `JUMPI` instruction. If the second parameter is non-zero, the program jumps to the *if* branch; otherwise, it proceeds to the *else* branch. As the concrete values of the instruction parameters cannot be known statically, we use symbolic variables to represent them. To transform this initial constraint into a state-related constraint, we designate the symbolic variables in this constraint as the trace-back target (Line 2) and collect their relationships with state variables during the backward traversal of instructions (Lines 5-10). In Line 7, we reconstruct the EVM stack by analyzing instructions in reverse order. Utilizing the recovered stack information, we generate new constraints derived from the initial constraint, subsequently refreshing the trace-back targets in a last-in-first-out (LIFO) manner in Line 8. This iterative process continues until all traced targets manifest as either state variables or state-independent constants, yielding a set of state-related constraints.

3) Example for State Identification: In Figure 3, the left side illustrates a simplified contract CFG with a determined code target, denoted as node T. As described in Algorithm 2, there are two paths reaching the target, i.e., A-B-E-T and A-C-F-T, with branch points at nodes A, B, E, and F. Backward analysis on these branch points yields constraints C_{a1}, C_{a2}, C_b, C_e, and C_f. The state constraints for the code target are then expressed as $(C_{a1} \cap C_b \cap C_e) \cup (C_{a2} \cap C_f)$. The right side of Figure 3 is a simplified bytecode segment of the branch point at node F, and Figure 4 demonstrates the backward analysis process at this branch point. In Figure 4, columns one and two display the program counter and corresponding instructions, while the third column represents the EVM stack information restored based on these instructions. The fourth column describes the current instruction's impact on stack elements, and the fifth column presents constraints generated during backward analysis. The sixth column displays elements in the trace-back list. Backward analysis initiates at the `JUMPI` instruction (pc =7), where the top of the EVM stack contains two elements, i.e., the jump destination `dst` and value `A`. The `JUMPI` instruction specifies that if $\mathtt{A} \neq 0$, pc jumps to dst; otherwise, it jumps to pc+1. Consequently, the first constraint, $\mathtt{A} \neq 0$, is derived, based on the jump direction, and the value A is added to the trace-back list. The `PUSH2` instruction (pc =6) pushes dst onto the stack, which does not affect the tracing target. Analyzing the `GT` instruction (pc =5) which compares the sizes of `B` and `C` and then assigns the result to `A`, produces a new constraint: $(\mathtt{A} = 1 \cap \mathtt{B} - \mathtt{C} > 0) \cup (\mathtt{A} = 0 \cap \mathtt{B} - \mathtt{C} <= 0)$. Consequently, the element in the trace-back list transitions from A to `B` and `C`. The subsequent `PUSH2` instruction (pc =4) reveals B as 5, and the `SLOAD` instruction (pc =3) indicates C is obtained from the current contract states at slot `D`. The backward analysis continues until the trace-back list is empty. These derived constraints constitute the state constraints for this branch point, denoted as C_f in Figure 3

B. Execution Trace Analysis

We analyze the execution trace of each seed during the fuzzing loop and employ a fitness metric algorithm to assess the proximity of the seeds to both code and state targets. This evaluation serves to measure the likelihood of a seed triggering a vulnerability. By utilizing this approach, we can strategically allocate testing resources to prioritize better seeds, thereby accelerating the testing process.

1) Code Distance Evaluation: The code target is essentially a program basic block in the contract's CFG. In this section, we refer to it as the target block. For a basic block BB, we define its distance to a target block TB as

$$d_{TB}(BB) = \begin{cases} d(BB, TB) & if\ BB \rightsquigarrow TB, \\ \infty & Otherwise. \end{cases} \tag{1}$$

where $BB \rightsquigarrow TB$ represents that TB is reachable from BB, and $d(BB, TB)$ represents the number of edges covered by the shortest path from BB to TB. Based on the above definition, for the block BB, we define its block distance $D(BB)$, as the harmonic mean of its distances to all target blocks, denoted as

$$D(BB) = \begin{cases} 0 & if\ BB \in \mathcal{Q}, \\ undefined & if\ (TB \in \mathcal{Q} \\ & |d_{TB}(BB) \equiv \infty), \\ |\mathcal{Q}| \times \left[\sum_{TB \in \mathcal{Q}} d_{TB}(BB)^{-1} \right]^{-1} & Otherwise. \end{cases} \tag{2}$$

where $\mathcal{Q}$ represents the set of target blocks and $|\mathcal{Q}|$ represents the size of $\mathcal{Q}$. Compared to the arithmetic mean, the harmonic mean distinguishes basic blocks that are closer to specific targets, and the more targets a block can reach, the smaller its

PC	Instruction	Stack (before instruction)	Description	Constraints	Trace Back Targets
7	JUMPI	top → dst, A, ...	Jump to dst	$C_f \leftarrow (A \neq 0)$	A
6	PUSH2 dst	top → A, ...	Push jump destination	$C_f \leftarrow C_f$	A
5	GT	top → B, C, ...	Greater-than comparison, i.e., A = GT(B,C)	$C_f \leftarrow C_f \cap (A = 1 \cap B - C > 0 \cup A = 0 \cap B - C <= 0)$	A ↗ B, A ↘ C
4	PUSH2 0x5	top → C, ...	Push 0x5, i.e., B = PUSH2(0x5)	$C_f \leftarrow C_f \cap B = 5$	A ↗ ~~B~~, A ↘ C
3	SLOAD	top → D, ...	Load from Storage, i.e., C = SLOAD(D)	$C_f \leftarrow C_f \cap C = Storage(D)$	A ↗ ~~B~~, ~~C~~ → D
2	PUSH2 0x0	top → ...	Push 0x0, i.e., D = PUSH2(0x0)	$C_f \leftarrow C_f \cap D = 0$	End

Fig. 4. Example for Backward Analysis at the branch point F.

distance will be, which facilitates covering as many targets as possible. After a seed S is executed, we analyze its execution trace and compute its code distance. We define the seed's code distance as the average of the smallest n block distances along the seed's execution trace, denoted as

$$CodeDistance(S) = \frac{1}{n} \times \sum_{BB \in \mathcal{N}} D(BB) \tag{3}$$

where $\mathcal{N}$ represents the set of n basic blocks with the smallest block distances along the execution trace. The value of n is contingent on the number of basic blocks. Notably, we avoid using the average of all block distances as the seed's code distance. Because the global optimal strategy can lead to discrepancy when there are multiple targets, where seeds reaching the target may exhibit larger distances than those that do not reach any target [16]. We also avoid using the smallest block distance as the seed's code distance, as it fails to effectively differentiate between different seeds.

2) State Distance Evaluation: Each code target corresponds to a specific state target, which is a collection of value ranges for state variables. For a seed S, we define its distance to a state target ST as

$$d_{ST}(S) = \begin{cases} 0 & if\ ST == \emptyset, \\ \sum_{SV \in \mathcal{M}} R(SV, ST) & Otherwise. \end{cases} \tag{4}$$

where SV represents the state variable, $\mathcal{M}$ represents all state variables involved in ST. $R(SV, ST)$ represents the distance of the value SV to the range ST throughout the entire execution trace. If SV falls within the range ST, $R(SV, ST)$ is considered to be 0. Based on the above definition, for seed S, we define its state distance as the harmonic mean of its distances to all state targets, denoted as

$$StateDistance(S) = \begin{cases} 0 & if\ (\exists ST \in \mathcal{P} \mid d_{ST}(S) == 0), \\ |\mathcal{P}| \times \left[\sum_{ST \in \mathcal{P}} d_{ST}(S)^{-1} \right]^{-1} & Otherwise. \end{cases} \tag{5}$$

where $\mathcal{P}$ represents the set of state targets, and $|\mathcal{P}|$ represents the size of $\mathcal{P}$.

3) Fitness Metric: We evaluate the potential of a seed to trigger vulnerabilities during execution by assigning a score that integrates both its code distance and state distance. It is defined as

$$D = \alpha \times \mathcal{F}(CodeDistance) + (1 - \alpha) \times \mathcal{F}(StateDistance) \tag{6}$$

$$SC_{bug} = (D + \beta)^{-1} \tag{7}$$

where α and β are constants less than one, and $\mathcal{F}$ is a normalization function to eliminate the difference in magnitude between the code distance and state distance. If a seed consists of multiple transactions, the score for the seed is determined by the transaction with the highest score. The fitness metric also incorporates feedback from branch coverage and data dependencies which are commonly used in graybox fuzzing, resulting in the following representation

$$Fitness = \gamma \times SC_{bug} + (1 - \gamma) \times (SC_{branch} + SC_{dep}) \tag{8}$$

where γ is a constant less than one, used to balance the weight between our vulnerability-directed strategies and conventional strategies, which focus on code coverage and the read/write frequency of state variables in the seed fitness metric. SC_{branch} is computed based on the count of newly covered branch edges. SC_{dep} is computed based on the frequency of state variable writes, which reflects how often state variables are modified during execution. This is important because frequent modifications to state variables can indicate areas of the contract that are more likely to impact its behavior and potentially expose vulnerabilities.

C. Fuzzing Loop

During the fuzzing loop stage, we use a genetic algorithm [37] guided by our proposed fitness metric for seed scheduling. We execute contracts using an independent EVM, which we instrument to manage the persistent storage of contract states and provide execution traces for subsequent analysis.

1) Seed Initialization: The testing seeds mainly consist of a *function selector*, *calldata*, and *value*. The *function selector* determines the function to be invoked and is computed using the contract ABI. At the beginning of the test, we initialize

two seeds for each function of the contract. *Calldata* refers to the function parameters. We first determine the parameter type according to the ABI. For fixed data types, such as `uint256`, we randomly select values within the valid input range. For non-fixed data types, like `string`, we determine a positive number as the data length and generate an input of that length. *Value* represents the funds transferred into the contract by the transaction. If the function is marked as payable, we attach an appropriate value to the transaction; otherwise, the *value* is 0.

2) Selection: We employ the aforementioned fitness metrics to select valuable seeds and allocate more testing resources to them, steering the fuzzing process toward vulnerabilities. Specifically, it adjusts the seed selection strategy by manipulating the probability of a seed being chosen. Seeds with higher fitness are more likely to be selected for subsequent crossover and mutation. In a fuzzing generation $\mathcal{G}$, we define $P(S)$ as the probability of selecting a seed S. It is computed as

$$P(S) = Fitness(S) / \sum_{G \in \mathcal{G}} Fitness(G). \tag{9}$$

3) Crossover: The initial seed consists of a single transaction. We use crossover to generate transaction sequences. During crossover, if the execution traces of two selected transactions exhibit a read-write dependency on the same state variable, we combine them in Read-After-Write (RAW) order to create a transaction sequence. If no dependency exists, the selected transactions are combined with a certain probability. Transaction sequences can be combined to generate longer sequences.

4) Mutation: The calldata of the seeds is mutated either randomly or from a mutation pool with a certain probability P. The mutation pool consists of state targets we identified and values that appeared in previous transactions, and it is continuously expanded during execution. This heuristic method increases the likelihood of passing narrow conditional branches.

5) Test Oracle: We define a vulnerability as being discovered by analyzing the execution trace to see if it violates the test oracles. These oracles are adapted from prior work, including ContractFuzzer [21] and Smartian [10], with modifications to enhance detection of particular vulnerability patterns. The following detection oracles are integrated into our approach.

The *Locking Ether* oracle consists of two sub-oracles. The first, *NoEtherSending*, ensures that the execution trace lacks any `SELFDESTRUCT` or `SUICIDE` instructions, as well as any `CALL` or `CREATE` instructions where *callvalue >0.* The second, *EtherAcceptance*, verifies that the contract can receive Ether by confirming the presence of a transaction with *callvalue >0* that ends with a `STOP` instruction.

The *Controlled/Dangerous Delegatecall* oracle is defined based on three conditions. First, the execution must include a `DELEGATECALL` instruction where the callee or arguments are tainted by `CALLDATALOAD`. Second, the execution must terminate with a `STOP` instruction, signaling normal termination. Third, the caller must be the designated attacker account.

The *Block Dependency* oracle consists of three conditions. First, the execution must contain comparison instructions (`LT`, `GT`, `SLT`, `SGT`, or `EQ`) with block-related oprands (`BLOCKHASH`, `COINBASE`, `TIMESTAMP`, `NUMBER`, `DIFFICULTY`, or `GASLIMIT`). Second, the execution should include a `CALL` instruction with *callvalue >0* or other sensitive instructions, such as `STATICCALL`, `SELFDESTRUCT`, `SUICIDE`, `CREATE`, or `DELEGATECALL`. Third, the execution must end with a `STOP` instruction.

The *Reentrancy* oracle is defined based on three conditions. The first condition checks for the presence of an `SLOAD` instruction with the oprand tainted by `CALLDATALOAD` or CALLER. The second condition verifies the presence of a `CALL` instruction with gas limit >2300 and a positive *callvalue* or tainted by `CALLDATALOAD` or CALLER. The third condition requires that an `SSTORE` instruction that modifies the same variable loaded in the first condition.

The *Ether Leaking* oracle is defined based on three conditions. First, the contract must transfer Ether to the attacker account. Second, the attacker has not previously sent Ether to the contract Third, the attacker's address has not been provided as an argument by non-attacker accounts.

The *Suicidal* oracle is composed of three conditions. First, the execution trace must include a `SELFDESTRUCT` instruction. Second, the caller must be the attacker account. Third, the attacker's address has not been passed as an argument by any non-attacker accounts.

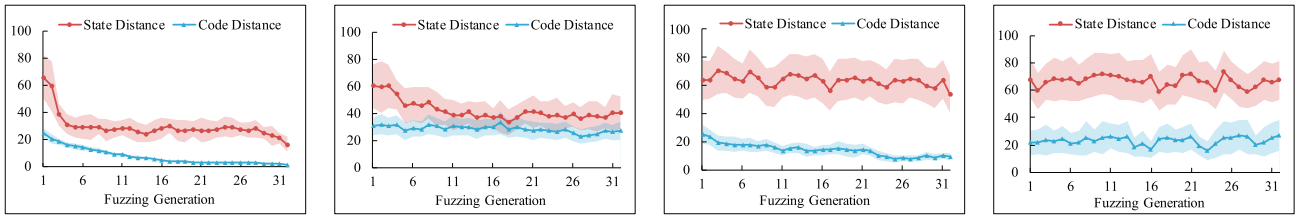
VI. Implementation

VULSEYE is implemented in Python, and we use py-evm [38], the official Python implementation of EVM, for the execution of smart contracts. We implement the static analysis phase of VULSEYE in an over-approximate way, so that more targets can be tested during the fuzzing phase, reducing the false negatives. Our fuzzing engine is built upon ConFuzzius [9]. Due to the uncertainty of the interactive objects when contracts initiate external calls, ensuring the success of every external call during testing is challenging [39]. To address this, we instrument py-evm to ensure that external calls within contracts consistently return appropriate values, thus preventing execution failures and facilitating cross-contract testing. We assign α in equation 6 a value of 0.5, determined through experimental validation, as it appropriately balances the weight of state distance and code distance in the seed score. We set β in equation 7 to 0.1, which controls the maximum value of SC_{bug} at 10, preventing excessive scores due to close distances and thus avoiding interference in the seed scoring. We set γ in equation 8 to 0.7, which achieves directed exploration while balancing coverage and the discovery of new paths, helping to avoid local optima traps. We modularize our test oracle to facilitate easy extension for supporting more vulnerability detections.

VII. Experiments

Rearch Questions: We conduct experiments to answer the following four questions. Our testing environment is comprised of a server with a 16-core Intel(R)-Xeon(R)-Gold-5218 CPU @2.30 GHz, 340GB of RAM, and the Ubuntu 18.04 LTS operating system.

- **RQ1:** How effective is VULSEYE in guiding testing toward specific targets, and how significantly do the code and state targets contribute to its performance?



(a) Distance of VULSEYE (b) Distance of configuration A (c) Distance of configuration B (d) Distance of configuration C

Fig. 5. Tendency of code/state distance on different ablation configurations.

- **RQ2:** How effective is VULSEYE in vulnerability detection compared to state-of-the-art fuzzers.
- **RQ3:** How does VULSEYE perform on code coverage?
- **RQ4:** How effective is VULSEYE compared to static tools?
- **RQ5:** How does VULSEYE perform on testing real-world smart contracts?

A. RQ1: Testing Specific Targets

We validate the efficacy of VULSEYE in testing specific code areas and create three ablations to evaluate the impact of both code and state targets that we introduced.

1) Dataset: We craft five benchmarks for RQ1 as existing datasets don't meet our needs and cannot contribute to a fair evaluation. We require: i) each contract to contain a vulnerability in a suitable branch to serve as a test target; ii) sequentially increasing difficulty in reaching these targets; iii) sufficiently challenging branch conditions to extend fuzzing time for observation. To achieve this, each contract is instrumented with hard-to-trigger branches by intentionally imposing restrictions on contract states. Specifically, each contract is designed with state-dependent conditions that are progressively harder to satisfy. We impose restrictions on state variables by requiring them to fall within narrow ranges. As the benchmarks progress, we increase both the number of functions and state constraints, causing the difficulty of reaching the target branch to rise exponentially. For example, consider a function with a condition involving 4 state variables. If each variable has a constraint range of [0-10] within a total value range of [0-100], the probability of randomly satisfying one state constraint is 0.1. To satisfy all 4 state constraints simultaneously, the probability is $(0.1)^4$. When the complexity increases to 6 state variables, the probability of satisfying all the conditions drops further to $(0.1)^6$. From these branches within each contract, we select one to inject a suicidal vulnerability as the testing target. Each benchmark consists of 10 contracts, and their average results are used to mitigate variability.

2) Ablation Study: We test VULSEYE on these five benchmarks, calculating the average time consumed to reach these targets. We create three ablation configurations, designated as A, B, and C. Configuration A excludes the guidance of code targets by removing the code distance evaluation component described in Section V-B1. Configuration B excludes the guidance of state targets by removing the state distance evaluation component described in Section V-B2. Configuration C excludes both code and state targets by removing both components. We conducted this experiment five times, and the results are presented in Table III. The first column denotes the five benchmarks, while the second to fifth columns display the average time taken by VULSEYE, configurations A, B, and C, respectively, to reach the targets. Columns six to eight present the speed-up factor achieved by VULSEYE. As depicted in Table III, VULSEYE achieves the shortest time to reach the targets in all five benchmarks compared to the ablations. Notably, the speed-up factor is more pronounced in the tests involving more intricate benchmarks, showcasing a more substantial efficiency improvement, reaching up to 8.46× when compared with configuration C. Moreover, configurations A and B take more time than VULSEYE (1.18 to 4.12×), suggesting that both code target and state target contribute to VULSEYE's ability in testing specific code areas.

TABLE III
COMPARISON OF TIME TO TARGETS

	T_V	T_A	T_B	T_C	$\frac{T_A}{T_V}$	$\frac{T_B}{T_V}$	$\frac{T_C}{T_V}$
B_1	**2.30**	4.35	2.71	4.14	1.89	1.18	1.80
B_2	**7.70**	17.79	9.32	18.25	2.31	1.21	2.37
B_3	**42.90**	98.67	110.25	138.57	2.30	2.57	3.23
B_4	**494.42**	1888.68	1112.45	2882.47	3.82	2.25	5.83
B_5	**1618.34**	6667.56	5502.36	13691.16	4.12	3.40	8.46

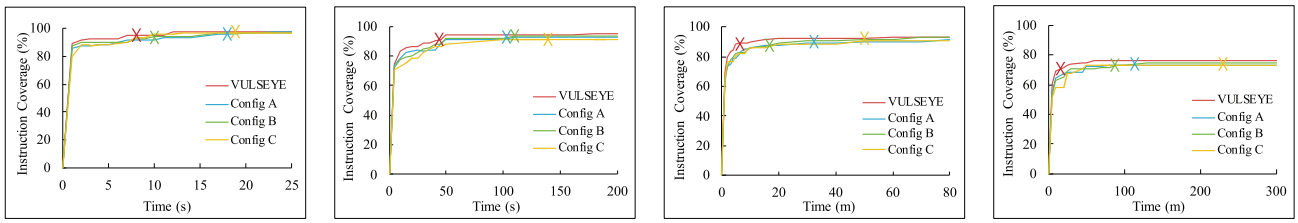
* B_1-B_5 are five benchmarks. T_V is the time (seconds) VULSEYE takes to reach targets, and T_A, T_B, T_C are the time of configuration A, B, and C.

To visually illustrate how VULSEYE guides testing toward these targets, we depict the tendency of code and state distance during the test of a contract within B_4 in Figure 5, with values averaged over all seeds within a fuzzing generation. In Figure 5a, we note a consistent decrease in the state and code distances of VULSEYE with the increase in fuzzing generations, exhibiting minimal standard deviations. In contrast, configurations A and B exhibit larger fluctuations and standard deviations in code distance and state distance, respectively. Configuration C, lacking guidance in both code and state space, exhibits random changes in both state and code distance in Figure 5d. Additionally, we depict the tendency of average code coverage across different benchmarks in Figure 6. It can be seen that VULSEYE outperforms the ablations in terms of coverage while reaching the targets faster.

Answer to RQ1: With the guidance of code and state targets we introduced, VULSEYE directs testing toward specific code areas effectively. It reaches challenging targets up to 8.4× faster than the baselines.

B. RQ2: Effectiveness of Vulnerability Detection

To answer RQ2, we conduct comparative experiments between VULSEYE and SOTA approaches on a ground-truth dataset consisting of 42 previously exploited projects.



(a) Code coverage of B_2 (b) Code coverage of B_3 (c) Code coverage of B_4 (d) Code coverage of B_5

Fig. 6. Tendency of code coverage on different benchmarks. The X indicates when and at what coverage the test target is reached.

1) Dataset: Our second dataset is derived from previously exploited projects curated by Durieux et al [40] and Torres et al [9]. However, these contracts lack the desired quantity and complexity for certain vulnerability types. To address this limitation, we have expanded and enriched these contracts to augment both their quantity and complexity. As a result, our dataset encompasses vulnerabilities of various types, including ether leaking (EL), block dependency (BD), reentrancy (RE), controlled delegatecall (CD), dangerous delegatecall (DD), suicidal (SD), and locking ether (LE).

2) Baselines Selection: In the experiment of RQ2, we compare VULSEYE to SOTA graybox fuzzers that are publicly available, including ConFuzzius [9], sFuzz [8], Smartian [10], IR-Fuzz [12], and ITYFuzz [14]. We choose CON FUZZIUS because it is the first hybrid fuzzer designed for smart contracts. We choose S FUZZ because its fuzzing strategy is inspired by AFL [41], a highly effective and widely-used fuzzer for C programs. We choose ITYFUZZ because, to our knowledge, it is the first tool to employ snapshot technology in smart contract fuzzing. We choose SMARTIAN and IR- FUZZ since they represent the latest advancements in research and assert superior performance compared to prior works. Like VULSEYE, these tools are designed to detect vulnerabilities in smart contracts and rely on source code to obtain the contract's ABI and perform static analysis. This static information is then used to guide fuzz testing. For example, SMARTIAN derives the read-write relationships of state variables from the source code to generate initial seeds, while IR- FUZZ analyzes *if* statements to allocate energy to the seeds.

3) Vulnerability Detection: We applied VULSEYE and SOTA tools on this ground-truth dataset. We performed five runs for each contract, employing a fuzzing timeout of 15 minutes for contracts with a code size below 200 lines and extending it to 30 minutes for larger contracts. The experimental results are presented in Table IV, where the first column indicates the contract, the second column specifies the vulnerability type within the contract, and the third column identifies the SWC ID according to the Smart Contract Weakness Classification [42]. The average time to detect vulnerabilities is recorded in columns four through nine.

As shown in Table IV, VULSEYE successfully detected 42 out of 42 vulnerabilities, CON FUZZIUS detected 38 vulnerabilities, and SMARTIAN detected 29 vulnerabilities. sFuzz and IR- FUZZ lacked test oracles for suicidal and ether leak, and exhibited poor performance in detecting reentrancy. As a result, they only detected 16 and 14 out of 42 vulnerabilities, respectively. ITYFuzz lacked test oracles for delegatecall

TABLE IV
COMPARISON OF TIME TO VULNERABILITIES

CID	BID	SWC-ID	VE	CF	SF	IF	IT	ST
1	EL	SWC-105	16.87	3.56	-	-	**1.02**	7.70
2	EL	SWC-105	**6.63**	26.10	-	-	10.30	900.00
3	EL	SWC-105	13.87	**2.24**	-	-	2.25	4.96
4	EL	SWC-105	**2.86**	15.39	-	-	3.63	900.00
5	EL	SWC-105	**0.15**	4.02	-	-	1.11	30.12
6	EL	SWC-105	**23.8**	185.60	-	-	32.95	1800.00
7	BD	SWC-120	**0.64**	4.21	10.38	55.50	0.90	6.57
8	BD	SWC-120	**1.01**	2.33	900.00	900.00	900.00	4.79
9	BD	SWC-120	**1.99**	64.50	5.50	5.63	900.00	18.94
10	BD	SWC-120	**0.11**	3.23	15.37	77.68	900.00	13.61
11	BD	SWC-120	9.15	32.11	5.89	369.22	**4.56**	68.42
12	BD	SWC-120	**1.06**	37.24	355.14	488.45	1800.00	539.67
13	RE	SWC-107	**2.05**	2.14	21.33	9.12	4.55	200.50
14	RE	SWC-107	**6.27**	15.02	900.00	900.00	9.03	87.43
15	RE	SWC-107	**16.89**	26.33	900.00	900.00	900.00	109.11
16	RE	SWC-107	**3.06**	8.12	900.00	900.00	19.48	900.00
17	RE	SWC-107	**8.73**	23.45	1800.00	1800.00	1800.00	55.34
18	RE	SWC-107	**211.29**	377.72	1800.00	1800.00	1800.00	1800.00
19	CD	SWC-112	5.45	3.30	5.29	24.06	-	**2.23**
20	CD	SWC-112	4.12	**2.50**	5.60	19.23	-	5.28
21	CD	SWC-112	**42.21**	224.34	900.00	476.45	-	48.33
22	CD	SWC-112	99.09	7.49	**5.12**	64.22	-	6.69
23	CD	SWC-112	**3.06**	69.68	10.88	343.78	-	11.61
24	CD	SWC-112	**5.45**	29.97	35.43	1800.00	-	26.78
25	DD	SWC-112	**3.92**	6.89	5.67	900.00	-	900.00
26	DD	SWC-112	**0.15**	2.02	5.30	13.10	-	2.29
27	DD	SWC-112	**126.59**	900.00	900.00	900.00	-	900.00
28	DD	SWC-112	24.53	34.23	5.98	**4.54**	-	5.32
29	DD	SWC-112	**1.25**	53.30	5.70	3.20	-	2.71
30	DD	SWC-112	**12.13**	31.54	30.31	1800.00	-	25.42
31	SD	SWC-106	**1.01**	2.21	-	-	1.33	1.88
32	SD	SWC-106	1.42	2.09	-	-	**0.89**	1.34
33	SD	SWC-106	**0.49**	4.18	-	-	1.33	6.61
34	SD	SWC-106	**4.47**	14.71	-	-	5.78	5.92
35	SD	SWC-106	**10.54**	19.66	-	-	31.26	226.45
36	SD	SWC-106	**107.29**	382.64	-	-	198.14	159.03
37	LE	SWC-132	✓	✗	✗	✗	✗	✗
38	LE	SWC-132	✓	✓	✗	✗	✗	✗
39	LE	SWC-132	✓	✗	✗	✗	✗	✗
40	LE	SWC-132	✓	✓	✗	✗	✗	✗
41	LE	SWC-132	✓	✓	✗	✗	✗	✗
42	LE	SWC-132	✓	✗	✗	✗	✗	✗

* VE represents VULSEYE, CF represents CONFUZZIUS, SF represents SFUZZ, IF represents IR-FUZZ, IT represents ITYFUZZ, and ST represents SMARTIAN. A dash (-) indicates that the tool does not support detection of this type of vulnerability, while 900.00 and 1800.00 represent timeouts where the vulnerability was not detected. Since locking ether (LE) vulnerabilities are only reported upon the fuzzing process is terminated, we use ✓to represent successful detections and ✗to represent failed detections specifically for the LE.

types and lock Ether, identifying 17 vulnerabilities. In terms of the time consumed to discover vulnerabilities, VULSEYE outperforms the other tools significantly. Particularly, our tool achieves the shortest time for the detection of 34 out of the 42 vulnerabilities (account for **81%**). Notably, this efficiency improvement is more significant especially for contracts larger than 200 lines (13 out of 14 vulnerabilities, account for **93%**). This is due to VULSEYE's prioritization of testing resources in vulnerable areas.

Answer to RQ2: Vuls- eye outperforms SOTA tools in smart contract vulnerability detec- tion. It exploited all vulnerabilities in the 42 projects and exhib- ited the fastest detection

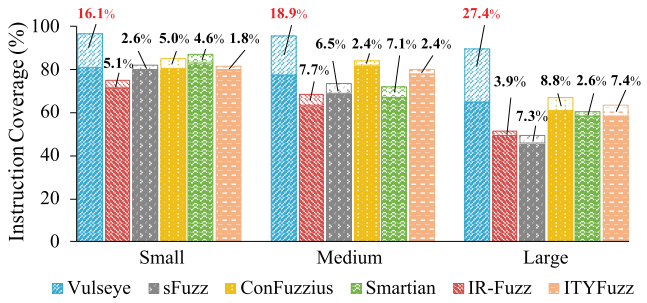


Fig. 7. Instruction coverage upon vulnerability discovery (dark shading) and upon fuzzing is terminated (dark shading and light shading).

time for 81% of them; this figure rises to 93% when the project is larger.

C. RQ3: Code Coverage

We categorized the contracts in RQ2 according to their size into three groups: small, medium, and large, and recorded the average instruction coverage (a form of code coverage) achieved by the six tools. As shown in Figure 7, scVulseye achieved the final instruction coverage of 96.9%, 95.5%, and 89.9%, surpassing all other tools. While achieving notable results in final instruction coverage, VULSEYE does not solely prioritize maximizing this metric. The light shading in Figure 7 represents the difference between the final instruction coverage and the coverage upon vulnerability discovery. A larger light shading area indicates a lower instruction coverage requirement for vulnerability detection. Comparing the light shading areas, they exhibit the largest proportion in VULSEYE (16.1%, 18.9%, 27.4%) within all three groups. This suggests that with the fuzzing strategy we employed, VULSEYE doesn't require achieving such a high code coverage to expose vulnerabilities, leading to a significant improvement in vulnerability discovery efficiency. It is worth noting that, although the code coverage at vulnerability discovery (dark shading) of comparison tools is lower than that of VULSEYE in Figure 7, it's due to their limitation in the notably lower final code coverage. However, their proportion of the dark shading areas remains higher than that of VULSEYE (the lower the better).

Answer to RQ3: Vuls- eye achieves the highest code coverage among the SOTA tools. It doesn't necessitate as high code coverage as SOTA tools to detect vulnerabili- ties, highlighting its significant advantage of prioritizing testing resources on vulnerable code and states.

D. RQ4: Effectiveness Compared to Static Tools

To further validate the effectiveness of our approach, which integrates vulnerability-prone zone identification with fuzzing, we compared VULSEYE against smart contract static analysis tools and examined to what extent our directed fuzzing reduces false positives. We first collect the vulnerable code areas identified on the RQ2 dataset by VULSEYE's vulnerability-prone zone identification (denoted as V- S). Next, we gather several leading static analysis tools: Slither [31], Smartcheck [43], Mythril [44], Securify [45], and Oyente [33], to scan the same dataset.

TABLE V
COMPARE VULSEYE WITH STATIC TOOLS

BID	SLITHER		SMARTCHECK		MYTHRIL		SECURIFY		OYENTE		V-S		VULSEYE	
	#	fp	#	fp	#	fp	#	fp	#	fp	#	fp	#	fp
EL	23	74%	0	N/A	0	N/A	0	N/A	0	N/A	8	25%	6	0%
BD	25	76%	21	71%	34	82%	30	80%	26	77%	13	54%	6	0%
RE	17	65%	23	74%	30	80%	15	60%	11	45%	12	50%	6	0%
CD	21	71%	0	N/A	19	68%	0	N/A	0	N/A	14	57%	6	0%
DD	0	N/A	0	N/A	0	N/A	0	N/A	0	N/A	16	63%	6	0%
SD	6	0%	0	N/A	6	0%	3	0%	0	N/A	6	0%	6	0%
LE	3	0%	4	0%	3	0%	4	0%	0	N/A	6	0%	6	0%
OT	39	100%	47	100%	14	100%	12	100%	22	100%	0	N/A	0	N/A
Total	134	69%	95	56%	106	60%	64	34%	59	29%	75	44%	**42**	**0%**

Table V summarizes the number of vulnerabilities identified by these tools along with their false positive rates. "OT" refers to other vulnerability types that fall outside the scope of our research. Columns 2-6 show that the five static tools detect only a subset of the six vulnerability types, and their reports contain a substantial number of false positives. For most vulnerability types, their false positive rate exceeds 60%, with the highest reaching 82%. In column 7, V- S, the vulnerability-prone zone identification component of VULSEYE, covers all 42 vulnerabilities. Since we followed an over-approximation principle to avoid prematurely excluding potentially vulnerable areas before fuzzing, V- S generates about 50% more reports for each vulnerability type. As shown in the final column, after applying the directed fuzzing in VULSEYE, the real vulnerabilities are accurately detected within these reports, and the false positive rate is reduced to 0.

Answer to RQ4: Bene- fiting from the hybrid approach, VULSEYE com- bines fast, comprehensive identification of vul- nerable code areas with a significantly lower false positive rate compared to static analysis tools.

E. RQ5: Performance on Real-World Contracts

To answer RQ5, we employed VULSEYE on 42,738 real-world smart contracts for bug detection and compared it with other tools. Additionally, we collected top 50 DApps from Ethereum for testing and found previously unknown bugs.

1) Finding Bugs in 42,738 Real-World Smart Contracts: We obtained the dataset from the study of Durieux et al [40], consisting of 47,587 real-world smart contracts extracted from Ethereum. After excluding contracts that couldn't be compiled by our compiler, this dataset comprised 42,738 real-world contracts. The source code for all contracts in this dataset is available on Etherscan [46].

We applied VULSEYE and other comparison tools on these contracts. We set the maximum number of test cases for each smart contract as 2,000, which we consider sufficient for most exploits. To further evaluate the performance of VULSEYE, we examine the identified vulnerable contracts manually to see whether they are true positives or not. Due to the large number of reported vulnerabilities, we are unable to check all of them. Instead, for each tool, we randomly sample 350 vulnerable contracts for manual verification (50 contracts for each vulnerability type). In cases where the reported results involve less than 50 contracts, all of them will be subject to manual verification. Table VI reveals that VULSEYE identified 4,845 vulnerabilities, surpassing other tools by up to 9.7×. Additionally, the overall true positive rate for VULSEYE in manual checking is the highest. Specifically,

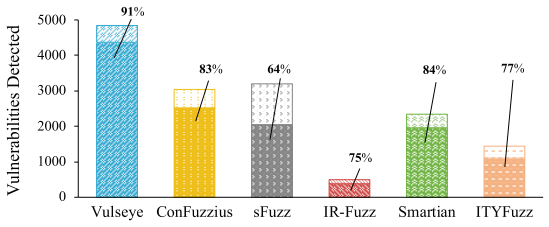


Fig. 8. The total number of identified vulnerabilities reported by each method (dark shading and light shading) and the true positive rates obtained from manual inspection (dark shading).

TABLE VI
COMPARISON OF DETECTED VULNERABILITIES AND TRUE POSITIVE RATE

BID	ITYFuzz		ConFuzzius		sFuzz		IR-Fuzz		Smartian		Vulseye	
	#	tp	#	tp	#	tp	#	tp	#	tp	#	tp
EL	838	78%	351	86%	0	N/A	0	N/A	387	86%	408	88%
BD	455	84%	2032	78%	2737	100%	343	84%	1825	80%	3720	100%
RE	132	64%	397	82%	278	78%	99	86%	25	68%	426	92%
CD	0	N/A	5	100%	36	33%	10	70%	1	0%	12	100%
DD	0	N/A	7	100%	66	42%	23	48%	8	100%	26	100%
SD	22	86%	131	80%	0	N/A	0	N/A	94	92%	87	100%
LE	0	N/A	113	84%	92	58%	25	60%	0	N/A	166	64%
Total	1447	77%	3036	83%	3209	64%	500	75%	2340	84%	**4,845**	**91%**

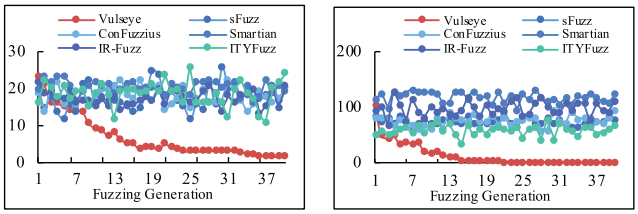
```
1 if(acc.balance>=MinSum && acc.balance>=_am && now>acc.
      unlockTime){
2   if(msg.sender.call.value(_am)()){acc.balance-=_am;} }
```

Fig. 9. Code Snippet of *THE_BANK*.

VULSEYE detected more vulnerabilities in 4 out of 7 vulnerability categories and demonstrated the highest true positive rate among the 6 vulnerability categories. Figure 8 provides an overview of the experimental results, clearly demonstrating that VULSEYE outperforms other tools in both vulnerability discovery quantity and true positive rate.

2) Case Study: To provide further insight on how the fuzzing strategies implemented in VULSEYE enhance bug detection and surpass existing fuzzers, we present a case study of a contract named *THE_BANK*,[1] which contains a reentrancy vulnerability. This vulnerability was detected by VULSEYE within the aforementioned real-world dataset but remained undiscovered by other tools. As shown in Figure 9, to trigger the reentrancy in Line 2, specific values of the state variables `acc.balance` and `acc.unlockTime` are required, which are influenced by other functions. Triggering the reentrancy vulnerability is challenging due to several factors. Firstly, multiple complex conditions must be satisfied to reach the external call that enables reentrancy. Secondly, the relevant state variables are influenced by other functions, requiring the attacker to carefully manipulate these values through intricate transaction sequences. Without state feedback and targeted guidance, it is difficult for a fuzzer to invoke the necessary functions to place the contract in the correct state and trigger the vulnerable external calls. To demonstrate how VULSEYE detects this contract vulnerability while other tools cannot, during testing of *THE_BANK* by VULSEYE and other fuzzers, we recorded the average code and state distances to the reentrancy vulnerability in each fuzzing generation, as depicted in Figure 10. VULSEYE showed a consistent decrease in both code and state distances, uncovering the vulnerability by the 37th test generation. It demonstrated that VULSEYE

[1]0xCB6fe98097fE7D6E00415Bb6623D5fC3EFFA4E83.



(a) Code Distance (b) State Distance

Fig. 10. Tendency of code/state distance during testing of *THE_BANK*.

successfully identified this vulnerable location, determined the necessary state conditions, and set corresponding targets to guide the fuzzing. In contrast, other fuzzers exhibited erratic changes in distance and failed to detect the vulnerability in the limited time. Their inefficient allocation of testing resources hinders their ability to set the contract state appropriately and exploit the vulnerability.

3) False Positives: During manual verification, we summarized several reasons for false positives reported by VULSEYE. For ether leaking (EL), the false positives attribute to contracts designed for lotteries, where the outward sending of ether is the original intention of the contract designer and should not be considered a vulnerability. For reentrancy (RE), the false positives result from instances where contracts are immune to reentrancy attacks due to inherent reentrancy protection or restrictions on external calls. In fact, it is non-trivial to automatically conduct reentrancy attacks on smart contracts. False positives in locking ether (LE) stem from contracts that cannot directly send funds but instead employ delegatecall to engage other contracts in fund transfers.

4) Finding 0-Day Bugs in Top DApps: We collected the top 50 popular DApps listed on BitDegree [47], a famous Web3 platform. These DApps primarily include projects in DeFi, gaming, and gambling, involving millions of transactions on the blockchain, and we utilized VULSEYE to test them. After five runs for each DApp, VULSEYE reported a total of 16 warnings involving 6 projects. After manual verification, we identified 11 previously unknown bugs related to reentrancy, controlled delegatecall, and block dependencies within 4 projects. The estimated balance of these four projects on BitDegree is about 2,500,000 USD, and these vulnerabilities put these funds at risk. We have responsibly reported these vulnerabilities to the corresponding authorities.

Answer to RQ5: VULSEYE is effective at identi- fying vulnerabilities in real-world scenarios, out- performing existing tools by reporting more vulner- abilities and achieving the highest true positive rate.

F. Threats to Validity

First, the performance of VULSEYE varies with the choice of the initial seeds. To ensure fairness, we used the same initial seeds in our comparative experiments with SOTA works, but other initial seeds or benchmarks may produce different results. We have open-sourced VULSEYE to facilitate further evaluation in other studies. Second, the false positive rate in RQ5 may not be entirely accurate. Due to the large number of contracts, we randomly selected 350 contracts from the positive results and manually checked them to estimate the tool's false positive rate for the entire dataset. Another threat

to validity relates to the type of vulnerabilities reported by ITYFUZZ in the comparative experiment. Since ITYFuzz's outputs do not directly indicate the vulnerability type, we determine it by manual inspection.

VIII. Related Work

We briefly review related works on smart contract vulnerability discovery and techniques of directed-graybox fuzzing.

A. Smart Contracts Vulnerability Discovery

1) Smart Contracts Fuzzing: VULSEYE is closely related to work on smart contracts fuzzing. Existing approaches [7], [8], [9], [11], [21], [48] have proven the effectiveness of fuzzing in exploiting vulnerabilities of smart contracts. ContractFuzzer [21] is the first black-box fuzzer for Ethereum smart contracts. Harvey [7] is a graybox fuzzer that applies the input-prediction technique to cover code that is guarded by narrow checks. sFuzz [8] combines the strategy of AFL fuzzer and a multi-objective adaptive strategy targeting those hard-to-cover branches in smart contracts. Su et al. [11] propose a reinforcement learning-guided fuzzer using code coverage and vulnerability discovery as rewards. EthPloit [49] combines static taint analysis, a dynamic seed strategy, and an instrumented Ethereum Virtual Machine to accurately simulate blockchain behaviors and generate exploits. ConFuzzius [9] leverages constraint solving to generate inputs that satisfy complex conditions and generates meaningful sequences of inputs at runtime using dynamic data dependency analysis. Most of these existing approaches are coverage-guided fuzzers that may waste testing resources on benign code areas and lack guidance in exploring the contract's state space. By contrast, VULSEYE is the first directed graybox fuzzer for smart contracts that concentrates testing resources on vulnerability-prone areas and performs directed fuzzing in both contract code and state spaces.

2) Smart Contracts Analysis: VULSEYE is related to work on smart contracts static analysis. Static analysis plays a crucial role in ensuring the security [50] and reliability [51] of software. Numerous static analysis frameworks for smart contracts have been proposed [31], [45], [52], [53]. Zeus [52] is a symbolic model checking framework for verification of smart contract correctness and fairness policies. Securify [45] analyzes smart contracts using a dependency graph to check compliance and violation patterns that capture sufficient conditions for proving if a property holds or not. Slither [31] is a widely used tool that integrates IR transfer and flow analysis. In our work, we use SlithIR as the foundation of static analysis, to identify code targets prior to conducting fuzz testing.

B. Directed-Graybox Fuzzing Techniques

VULSEYE is closely related to work on directed-graybox techniques [17], [18], [54]. AFLGo [17] is one of the first directed-graybox systems. It utilizes a simulated annealing-based power schedule to drive the seed toward target sites. Hawkeye [18] boosts the speed for fuzzer to the target sites by utilizing power scheduling, adaptive mutation, and seed prioritization. Existing directed-graybox fuzzers cannot be used to detect vulnerabilities in smart contracts. Firstly, these tools lack the capability to autonomously identify vulnerable areas within smart contracts, relying instead on manual or external information to specify testing targets. Additionally, their exclusive focus on the code space makes them less effective in testing stateful programs such as smart contracts. VULSEYE addresses these challenges and achieves effective stateful directed fuzzing.

IX. Discussion

VULSEYE inherits some limitations which can be further improved in future works. Firstly, during the target state identification process, if a mapping uses a symbolic address as the key, it cannot be determined statically, and the complexity increases when symbolic addresses are derived from another mapping or array [55]. Therefore, we exclude symbolic addresses from consideration and focus on measuring distances for determinable storage slots. To address this limitation, future work could introduce pointer analysis techniques to improve the handling of symbolic addresses.

Secondly, like other smart contract fuzzers, VULSEYE relies on predefined test oracles to determine whether vulnerabilities are triggered However, the existing test oracles may not be applicable to some new types of vulnerabilities or their variants, which could result in missed detections. Although we have modularized the oracle to allow dynamic updates and adjustments, expert knowledge and manual intervention are still required to optimize and extend its applicability. A potential future solution could involve the capabilities of large language models to automate the creation of test oracles, helping to address this challenge and improve oracle adaptability.

Additionally, akin to other fuzzers, VULSEYE aims for completeness in its analysis rather than soundness. As a result, VULSEYE cannot guarantee the absence of false negatives.

X. Conclusion

In this paper, we propose VULSEYE, a stateful directed fuzzer for smart contracts. The key idea is to enhance fuzzing efficiency by exploring critical contract states through feedback from the contract state space and prioritizing testing resources to vulnerable code areas. To achieve this, we introduce code targets and state targets to flag the vulnerable contract code areas and states. We design a novel fitness metric algorithm to steer the fuzzing toward these identified targets. Experimental results show that VULSEYE outperforms existing fuzzers and is effective in finding bugs in real-world scenarios.

Acknowledgment

Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors and do not reflect the views of the National Research Foundation Singapore or the Cyber Security Agency of Singapore.

References

[1] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 615–627.

[2] C. Staff. (2023). *What Was the Dao?*. [Online]. Available: https://www.gemini.com/cryptopedia/the-dao-hack-makerdao

[3] REKT. (2021). *Poly Network—Rekt*. [Online]. Available: https://www.gemini.com/cryptopedia/the-dao-hack-makerdao

[4] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 1–36, Jan. 2022.

[5] S. Wu et al., "Are we there yet? Unraveling the state-of-the-art smart contract fuzzers," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, Apr. 2024, pp. 1–13.

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

[7] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Nov. 2020, pp. 1398–1409.

[8] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "SFuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, Oct. 2020, pp. 778–788.

[9] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "ConFuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Sep. 2021, pp. 103–119.

[10] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "SMARTIAN: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2021, pp. 227–239.

[11] J. Su, H.-N. Dai, L. Zhao, Z. Zheng, and X. Luo, "Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Oct. 2022, pp. 1–12.

[12] Z. Liu et al., "Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 1237–1251, 2023.

[13] S. So, S. Hong, and H. Oh, "SmarTest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution," in *Proc. USENIX Secur. Symp. (USENIX Security)*, 2021.

[14] C. Shou, S. Tan, and K. Sen, "ItyFuzz: Snapshot-based fuzzer for smart contract," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2023, pp. 322–333.

[15] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, May 2022, pp. 1621–1633.

[16] P. Wang, X. Zhou, T. Yue, P. Lin, Y. Liu, and K. Lu, "The progress, challenges, and perspectives of directed greybox fuzzing," 2020, *arXiv:2005.11907*.

[17] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2329–2344.

[18] H. Chen et al., "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 2095–2108.

[19] H. Zheng et al., "FISHFUZZ: Catch deeper bugs by throwing larger nets," in *Proc. USENIX Secur. Symp. (USENIX Security)*, 2023, pp. 1343–1360.

[20] C. Luo, W. Meng, and P. Li, "SelectFuzz: Efficient directed fuzzing with selective path exploration," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 2693–2707.

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

[22] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: Foundations, design landscape and research directions," 2016, *arXiv:1608.00771*.

[23] G. Wood. (2022). *Ethereum: A Secure Decentralised Generalised Transaction Ledger Berlin Version*. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[24] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "SAILFISH: Vetting smart contract state-inconsistency bugs in seconds," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 161–178.

[25] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, Nov. 2008, pp. 151–166.

[26] K. Kim, S. Kim, K. R. B. Butler, A. Bianchi, R. Kennell, and D. Tian, "Fuzz the power: Dual-role state guided black-box fuzzing for USB power delivery," in *Proc. USENIX Secur. Symp. (USENIX Secur.)*, 2023, pp. 5845–5861.

[27] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018.

[28] W. Sun et al., "A survey of source code search: A 3-dimensional perspective," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 6, pp. 1–51, Jul. 2024.

[29] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *Proc. USENIX Secur. Symp. (USENIX Secur.)*, Jan. 2022, pp. 3255–3272.

[30] W. Sun et al., "Abstract syntax tree for programming language understanding and representation: How far are we?," 2023, *arXiv:2312.00413*.

[31] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, May 2019, pp. 8–15.

[32] M. Nachtigall, L. N. Q. Do, and E. Bodden, "Explaining static analysis—A perspective," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. Workshop (ASEW)*, Nov. 2019, pp. 29–32.

[33] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. Conf. Comput. Commun. Secur. (CCS)*, May 2021, pp. 1–3.

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

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

[36] (2023). *NetworkX*. [Online]. Available: https://networkx.org/

[37] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: Past, present, and future," *Multimedia Tools Appl.*, vol. 80, no. 5, pp. 8091–8126, Feb. 2021.

[38] Ethereum. (2023). *Py-evm*. [Online]. Available: https://github.com/ethereum/py-evm

[39] Y. Xue et al., "XFuzz: Machine learning guided cross-contract fuzzing," *IEEE Trans. Dependable Secure Comput. (TDSC)*, vol. 21, no. 2, pp. 515–529, Apr. 2024.

[40] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, Oct. 2020, pp. 530–541.

[41] Google. (2022). *American Fuzzy Lop*. [Online]. Available: https://github.com/google/AFL

[42] SmartContractSecurity. (2020). *Smart Contract Weakness Classification*. [Online]. Available: https://swcregistry.io/

[43] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of Ethereum smart contracts," in *Proc. IEEE/ACM 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, May 2018, pp. 9–16.

[44] ConsenSys. (2023). *Mythril*. [Online]. Available: https://github.com/ConsenSys/mythril/

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

[46] Etherscan. (2023). *Etherscan.IO*. [Online]. Available: https://etherscan.io/

[47] (2023). *BitDegree*. [Online]. Available: https://cn.bitdegree.org/

[48] M. Ye, Y. Nan, Z. Zheng, D. Wu, and H. Li, "Detecting state inconsistency bugs in DApps via on-chain transaction replay and fuzzing," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2023, pp. 298–309.

[49] Q. Zhang, Y. Wang, J. Li, and S. Ma, "EthPloit: From fuzzing to efficient exploit generation against smart contracts," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Feb. 2020, pp. 116–126.

[50] T. Han et al., "Mutual information guided backdoor mitigation for pre-trained encoders," 2024, *arXiv:2406.03508*.

[51] W. Sun, X. Wang, H. Wu, D. Duan, Z. Sun, and Z. Chen, "MAF: Method-anchored test fragmentation for test code plagiarism detection," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Softw. Eng. Educ. Training (ICSE-SEET)*, May 2019, pp. 110–120.

[52] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–12.

[53] Y. Sun et al., "GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, Apr. 2024, pp. 1–13.

[54] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "BEACON: Directed grey-box fuzzing with provable path pruning," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 36–50.

[55] C. Fang et al., "Esale: Enhancing code-summary alignment learning for source code summarization," *IEEE Trans. Softw. Eng.*, vol. 50, no. 8, pp. 2077–2095, Aug. 2024.