V2E: Validating Smart Contract Vulnerabilities through Profit-driven Exploit Generation and Execution

JINGWEN ZHANG, Sun Yat-sen University and Peng Cheng Laboratory, China
YUHONG NAN*, Sun Yat-sen University, China
KAIWEN NING, Sun Yat-sen University and Peng Cheng Laboratory, China
MINGXI YE, Sun Yat-sen University, China
WEI LI, Sun Yat-sen University, China
YUMING XIAO, Sun Yat-sen University, China
YUMING FENG, Peng Cheng Laboratory, China
WEIZHE ZHANG, Harbin Institute of Technology, China
ZIBIN ZHENG, Sun Yat-sen University, China

Smart contracts are a critical component of blockchain systems. Due to the large amount of digital assets carried by smart contracts, their security is of critical importance. Although numerous tools have been developed for detecting smart contract vulnerability, their effectiveness remains limited, particularly due to the high false positives included in the reported results. Therefore, developers and auditors are often overwhelmed with manually verifying the reported issues. A fundamental reason behind this is that while a reported vulnerability satisfies specific vulnerable patterns, it may not actually be exploitable, either because the vulnerable code cannot be triggered or it does not result in any financial loss.

In this paper, we propose *V2E*, a new framework for validating whether a reported vulnerability is truly exploitable. The core idea of *V2E* is to automatically generate executable Proof-of-Concept Exploit (PoC for short), and then assess if the vulnerability could be triggered and incur any real damage (i.e., causing financial loss) by the PoC. While LLMs have shown proficiency in PoC generation, achieving our task is by no means trivial. In detail, it is difficult for LLM to: (1) generate and update PoC to trigger a specific vulnerability, (2) evaluate the PoC's effectiveness to validate exploitable vulnerability. To this end, *V2E* automates the whole process through a novel combination of PoC generation, validation, and refinement: (1) Firstly, *V2E* generates targeted PoCs by analyzing potential vulnerability paths. (2) Then, *V2E* verifies the validity of PoCs through triggerability and profitability analysis. (3) In addition, *V2E* iteratively refines the generated PoC based on PoC execution feedback, therefore, increasing the chance to confirm the vulnerability. Evaluation on 264 manually labeled contracts shows that *V2E* outperforms the baseline approach. Particularly, *V2E* successfully identifies 102 out of 124 exploitable vulnerabilities, achieving a precision of 91.9% and a recall of 82.3%. In addition, it successfully eliminates 71 out of 140 false alarms (50.7%). Besides, *V2E* effectively enhances the performance of SOTA tools. It reduces the false positive rates of Slither by 76.9%, Mythril by 56.9% and Confuzzius by 65%.

*Yuhong Nan is the corresponding author.

Authors' Contact Information: Jingwen Zhang, Sun Yat-sen University and Peng Cheng Laboratory, China, zhangjw273@mail2.sysu.edu.cn; Yuhong Nan, nanyh@mail.sysu.edu.cn, Sun Yat-sen University, China; Kaiwen Ning, Sun Yat-sen University and Peng Cheng Laboratory, China, ningkw@mail2.sysu.edu.cn; Mingxi Ye, Sun Yat-sen University, China, yemx6@mail2.sysu.edu.cn; Wei Li, Sun Yat-sen University, China, liwei378@mail2.sysu.edu.cn; Yuming Xiao, Sun Yat-sen University, China, xiaoym23@mail2.sysu.edu.cn; Yuming Feng, Peng Cheng Laboratory, China, fengym@pcl.ac.cn; Weizhe Zhang, Harbin Institute of Technology, China, wzzhang@hit.edu.cn; Zibin Zheng, Sun Yat-sen University, China, zhzibin@mail.sysu.edu.cn.

ACM 2994-970X/2026/7-ARTFSE101
https://doi.org/10.1145/3808108

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

Additional Key Words and Phrases: Smart Contract, Vulnerability Validation, Proof of Concept

ACM Reference Format:
Jingwen Zhang, Yuhong Nan, Kaiwen Ning, Mingxi Ye, Wei Li, Yuming Xiao, Yuming Feng, Weizhe Zhang, and Zibin Zheng. 2026. V2E: Validating Smart Contract Vulnerabilities through Profit-driven Exploit Generation and Execution. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE101 (July 2026), 23 pages. https://doi.org/10.1145/3808108

1 Introduction

Smart contracts are self-executing code and form a critical component of blockchain systems. Due to their inherent financial properties, smart contracts have become prime targets for attackers. For example, the notorious DAO attack [35] resulted in losses exceeding 50 million USD. To secure smart contracts, numerous studies have proposed effective vulnerability detection techniques, such as static analysis [21, 30], fuzzing [27, 38], and machine learning based methods [34, 44].

However, prior studies [5, 23] have shown that existing smart contract vulnerability detection tools often suffer from high false positive rates, limiting their practical usability. More specifically, many of the reported vulnerabilities are not truly exploitable in real-world scenarios. One key reason is the absence of an appropriate execution context to trigger the vulnerability. Since these tools typically rely on static analysis or predefined patterns [18], they struggle to capture contextual information necessary for exploiting the vulnerability. Moreover, even when the vulnerability can be triggered, it could still be unexploited if it does not lead to any tangible damage. In other words, if a vulnerability can be successfully triggered but does not result in any profit, attackers are unlikely to exploit it, since their primary motivation is financial gain [67].

Therefore, helping developers and auditors automatically verify the reported vulnerabilities is critically important, as it significantly accelerates vulnerability response and remediation. To achieve this goal, one promising approach is to automatically generate the Proof-of-Concept Exploit (called PoC), a code snippet that is used to prove the validity of the reported vulnerability. In detail, PoCs provide both an executable script to trigger the vulnerability and concrete evidence of its impact. Besides, recent advancements [39, 46] have shown LLMs perform well in PoC generation.

Challenges. However, most smart contract vulnerabilities are state-dependent logic bugs, and the exploitation primarily results in inconsistency from intended execution rather than program crashes. This makes it difficult for LLMs to generate valid PoCs for smart contracts. Specifically, there are two main challenges:

• **Generating and refining vulnerability-specific PoC.** The reasoning capabilities of LLMs are insufficient for directly generating effective PoCs. On the one hand, most static analysis [13, 26] and machine learning methods [50, 60] typically can only pinpoint the locations of vulnerabilities (e.g., code segments). And LLMs are struggling to infer the correct exploit path from a large number of publicly available functions within a contract. On the other hand, LLMs cannot dynamically explore the contract's state space, involving intricate transaction sequences and parameter choices. Consequently, PoCs generated by LLMs may fail to trigger the vulnerability or even fail to execute.

• **Assessing effectiveness of PoC.** In the meantime, LLM themselves cannot determine whether a generated PoC is effective, since smart contract vulnerabilities can only be confirmed through actual execution. Moreover, using static analysis and fuzzing methods to evaluate PoC effectiveness is still challenging, as existing techniques [31, 53, 55] either cannot handle complex processes in PoC, such as exploit preparation, exploitation, and post-processing, or assess vulnerability severity under the obfuscation introduced by PoC characteristics, like cheatcodes. In addition, recent work (i.e., A1 [17]) adopts inline commands (e.g., `forge test`) to obtain PoC outcomes. Unfortunately,

A1 offers very limited extensibility for PoC-centric vulnerability assessment, as it cannot capture fine-grained execution information such as opcode invocation sequences and dynamic state flow. To this end, validating and assessing smart contract PoCs remains quite difficult.

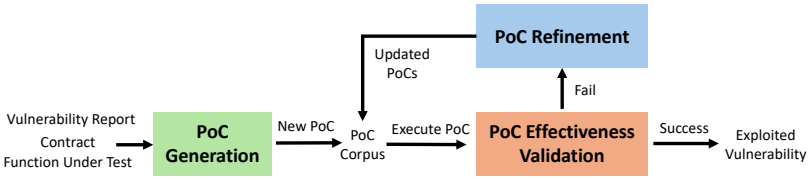


Fig. 1. An example of vulnerability validating process in *V2E*.

Our work. In this paper, we introduce *V2E*, a novel framework for validating smart contract vulnerabilities. As shown in Fig. 1, *V2E* takes vulnerability report, contract source code, and function under test as input and attempts to generate PoCs to assess vulnerability. To address the challenges above, *V2E* is designed as follows: (1) analyzing vulnerability path to synthesize tailored PoC; (2) verifying effectiveness of PoC through triggerability and profitability; (3) using execution feedback to guide PoC update.

Specifically, to synthesize a tailored PoC, *V2E* employs fine-grained static analysis to search for vulnerability entry points, such as publicly accessible functions, and constructs potential vulnerability paths. Then, by combining vulnerability characteristics and contract deployment requirements, *V2E* generates vulnerability-specific PoCs and stores them in PoC Corpus (Section 4.1). To verify the effectiveness of PoC, *V2E* builds a PoC effectiveness validation framework. The framework extracts a PoC from the corpus each time and executes it. Then, *V2E* analyzes whether the PoC can trigger vulnerabilities and assesses the severity of the vulnerabilities. If the PoC can both trigger the vulnerability and cause profit for attackers, *V2E* considers the vulnerability is exploitable (Section 4.2). Otherwise, *V2E* refines the PoC. To improve the opportunity of vulnerability validation, *V2E* combines execution feedback for vulnerability exploration enhancement. In the case of PoC execution failure, *V2E* revises the PoC using the semantics of the execution bytecode. Conversely, if the PoC runs successfully but does not trigger the vulnerability or yield profits, *V2E* leverages primitive operations to refine the PoC. Then, *V2E* puts the updated PoC back to the corpus for further analysis (Section 4.3).

To evaluate the effectiveness of *V2E*, we collect 264 vulnerable contracts from the SmartBugs dataset, including 64 manually constructed contracts and 200 real-world on-chain contracts. We manually annotate the exploitability of each vulnerability based on whether it could be triggered for profit. Experiment results show that *V2E* outperforms the LLM-based baseline method. In particular, *V2E* accurately identifies 102 in 124 exploitable vulnerabilities, achieving a precision of 91.9%, a recall of 82.3%, and correctly filters out 71 in 140 (50.7%) false alarms. Moreover, *V2E* effectively reduces the false positive rate of Slither by 76.9%, Mythril by 56.9% and Confuzzius by 65%.

In summary, this paper makes the following contributions:

- We highlight the need for vulnerability validation in smart contracts. We propose the idea to validate vulnerability based on triggerability and profitability.
- We propose *V2E*, a novel automated smart contract vulnerability validating framework, to generate and refine vulnerability-specific PoC, observe and interpret its execution results to confirm a reported vulnerability.
- We conduct extensive experiments to demonstrate the effectiveness of *V2E*. The results show that *V2E* can effectively evaluate the vulnerabilities and enhance the performance of existing detection tools.

The rest of the paper is organized as follows: Section 2 introduces the knowledge and motivation of our research. Section 3 presents the design choices and the workflow of *V2E*. Section 4 outlines technology details of *V2E*. Section 5 discusses the design and the results of experiments. Section 6 provides a discussion and the limitations of *V2E*. Section 7 reviews related research work. Section 8 concludes the paper, and Section 9 provides details on the data availability.

2 Background and Motivation

2.1 Background

Smart contracts and vulnerability detection. A smart contract [36] is a piece of code that runs automatically on the blockchain, like Ethereum [54]. To execute a smart contract, the contract must first be compiled into bytecode and deployed onto the blockchain. Due to the large amount of funds stored in smart contracts and their immutable nature once deployed, vulnerabilities in smart contracts are a major concern for both developers and attackers. Since most vulnerabilities in smart contracts are logic-based, different vulnerabilities exhibit distinct features, such as nested calls in Reentrancy, reliance on block attributes in Randomness from Chain Attributes, and inconsistent execution results caused by Transaction Order Dependence. Although many studies [3, 20, 38] have proposed vulnerability detection methods, the vast majority either cannot confirm whether a vulnerability is actually triggerable or fail to assess its real-world impact, resulting in false alarms.

Vulnerability validation. Compared to vulnerability detection, vulnerability validation focuses on assessing whether a reported vulnerability can be exploited, including being triggered under real-world conditions and causing real damage. In smart contracts, this typically requires analyzing whether the vulnerability can be triggered in an on-chain environment and whether the attacker can gain profit. Besides, vulnerability validation is crucial, as it helps developers prioritize and respond to harmful vulnerabilities more efficiently under limited resources, such as time and manpower.

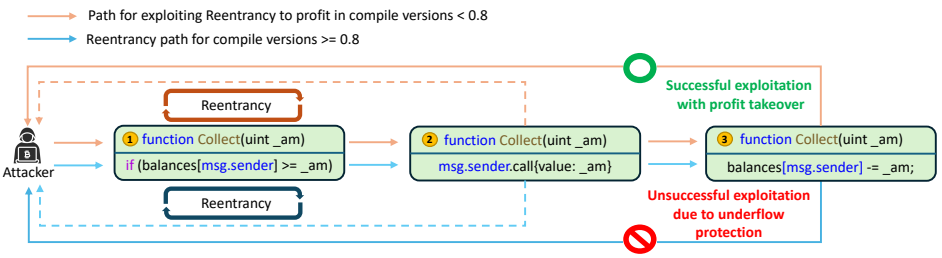


Fig. 2. Motivating example. Even for the same vulnerability, different compile versions lead to different exploitability.

2.2 Motivating Example

The function named `Collect` shown in Fig. 2 contains a Reentrancy vulnerability, where an attacker can hijack the control flow at Step 2 and repeatedly withdraw funds from the function. However, what makes this example interesting is that the exploitability of the vulnerability varies depending on the compiler version. Specifically, compile versions above 0.8 automatically insert underflow checks for all arithmetic operations during compilation [49]. As a result, in compile versions greater than 0.8, this function can no longer be exploited for profit via Reentrancy, and the Reentrancy can be considered mitigated. However, for compile versions below 0.8, the compiler does not insert additional safety checks, allowing Step 3 to execute successfully despite an underflow. The attacker can gain profit and pose a threat to the contract.

Motivation of *V2E*. Despite the numerous smart contract vulnerability detection tools available, accurately identifying the exploitable vulnerability shown in Fig. 2 is rather difficult. More specifically, we outline the following motivations:

• **Confirming the triggerability.** Most vulnerability detection tools ignore the execution context and often rely on code patterns to ensure completeness of their results. However, this approach is inherently fragile, as it may cause benign code to be misidentified. For example, even well-maintained tools, such as Slither [13], Mythril [37], and LLM-based methods, incorrectly identify the code in Fig. 2 as vulnerable across all compile versions, failing to account for execution differences introduced by varying compiler versions.

• **Evaluating exploit severity.** The severity of a detected vulnerability can only be confirmed by execution, as smart contracts are state-dependent. However, fuzzing-based approaches may misjudge the actual impact of vulnerabilities due to the low semantic nature of bytecode. For instance, at the bytecode level, `balances[msg.sender]` in Step 3 of Fig. 2 is an `id` determined at runtime. Existing methods, such as Confuzzius [51], detect the state change but are unable to interpret further what changes in `id`'s value signify. As a result, existing methods are unable to perform a fine-grained assessment of exploit severity.

Key observation for exploitable smart contract vulnerability . Through the analysis of past attacks and the results produced by vulnerability detection tools, we summarize the critical criteria that constitute exploitable smart contract vulnerability as follows:

(1) *Valid trigger.* Confirming that a vulnerability can be triggered in an on-chain environment is crucial, as the same code fragment may behave differently under varying execution contexts. For example, in Step 3 at Fig. 2, when `balances[msg.sender]` is less than `_am`, it will throw an error in compile versions above 0.8 due to built-in underflow checks, but will execute successfully in versions below 0.8. Such fine-grained behavioral differences can only be demonstrated by appropriate execution contexts.

(2) *Valid profit.* In smart contracts, evaluating the profitability of a vulnerability provides a more accurate reflection of its severity [27, 68]. On one hand, attackers, developers, and auditors are primarily concerned with vulnerabilities that impact the security of funds [42, 61]. On the other hand, a vulnerability may be triggerable but have no practical consequence. For example, in compile versions above 0.8 at Fig. 2, an attacker can trigger Reentrancy without breaking underflow protection as long as the total withdrawn amount is less than `balances[msg.sender]`. However, in such cases, the attacker gains no profit and is therefore unlikely to exploit in practice.

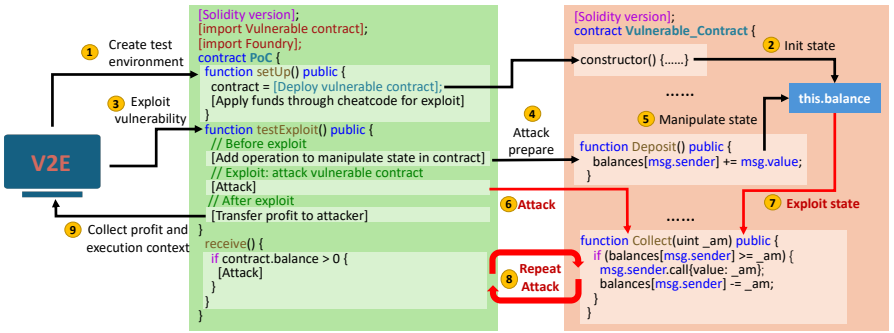


Fig. 3. The vulnerability validation process for motivating example in Fig. 2 by using PoC.

Our solution. To validate the exploitability of a vulnerability, *V2E* generates and validates the triggerability and profitability of PoCs based on the Foundry framework [15]. In detail, PoC is a

minimal example designed to demonstrate that a vulnerability can be practically exploited. In the context of smart contracts, a PoC is typically a contract that can interact with the target contract. Besides, among PoC development, Foundry is one of the most widely used frameworks, as it provides a complete testing environment and offers a range of cheatcodes. As shown in Fig. 3, *V2E* uses a PoC to set up the vulnerability testing environment (steps 1–2), and then proceeds to exploit the vulnerability (step 3). Specifically, it first manipulates critical contract states before the attack (steps 4–5) to simulate realistic conditions. Then, *V2E* launches the attack by invoking the target function (steps 6–8). After the attack is completed, *V2E* collects profit and execution-related information (step 9) to determine whether the PoC is effective.

We use the example in Fig. 2 to illustrate how *V2E* identifies the exploitable vulnerability. When *V2E* receives the code, it begins by analyzing the vulnerability path and generating a PoC that attempts to exploit Reentrancy. For compile versions below 0.8, *V2E* finds that the generated PoC successfully triggers the vulnerability and results in an increase in the attacker's balance. Therefore, *V2E* determines that the vulnerability is exploitable and outputs the result. For versions above 0.8, when *V2E* attempts to profit from the vulnerability, it observes that the PoC consistently fails to trigger Reentrancy during execution. In response, it iteratively analyzes the cause of the failure and updates the PoC. However, even when the PoC successfully triggers Reentrancy, *V2E* finds that the attacker's balance does not increase. As a result, it concludes that the vulnerable path is not exploitable and ultimately filters it out.

2.3 Scope of *V2E*

In this paper, *V2E* adopts a stricter standard for vulnerability evaluation. In detail, *V2E* considers a detected vulnerability to be valid only if it is both triggerable and profitable, as it can be exploited in the real world. Otherwise, *V2E* treats it as a false alarm.

The primary goal of *V2E* is to confirm exploitable smart contract vulnerabilities. Specifically, *V2E* aims to: (1) validate exploitable vulnerabilities that pose real profits within vulnerable contracts, and (2) eliminate false alarms introduced by existing methods. By doing so, *V2E* enables more efficient vulnerability response and remediation under constrained resources.

Currently, *V2E* focuses on five categories defined in the *Smart Contract Weakness Classification (SWC)* [47]: *Unprotected Ether Withdrawal (UEW), Unprotected Selfdestruct (US), Reentrancy (RE), Transaction Order Dependence (TOD)*, and *Randomness from Chain Attributes (RCA)*. The reasons are as follows: they are commonly observed SWC from real-world smart contract audits [69]. Besides, since not all SWC categories represent vulnerabilities with real-world damage, e.g., Code With No Effects, we selectively include only those that can potentially allow an attacker to gain profit when triggered.

3 Design of *V2E*

3.1 Key Ideas

Vulnerability Path Guided PoC Synthesis. Different vulnerabilities have different exploitation methods. An intuitive approach is to leverage the reasoning capabilities of LLMs to analyze vulnerable contracts and generate PoCs directly. However, PoC generation is a complex coding task, involving vulnerability understanding, exploit path searching, and code synthesis. LLMs lack such strong reasoning abilities [24].

To enable LLMs to generate valid PoCs, *V2E* employs fine-grained static analysis, including state dependency and access control analysis, to search for potential vulnerability entry functions. Prior research [68] shows that over 86% of exploits (including both inter- and intra-contract vulnerabilities) follow a "state-modify-then-state-read" pattern. Therefore, by analyzing the entry

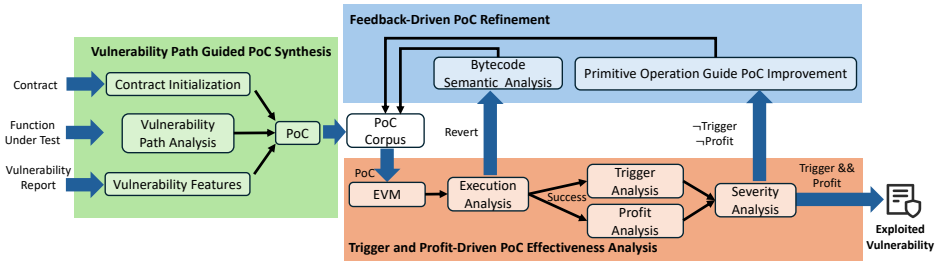


Fig. 4. The detailed workflow of *V2E*.

functions that affect the state reading in vulnerable functions, it is possible to build vulnerability exploit paths. Note that smart contracts often include access control mechanisms, such as restricting access to specific accounts or preventing direct external calls. Thus, access control analysis can reduce ineffective detections and false positives. Besides, *V2E* initializes the contract in the PoC before testing, enabling support for undeployed contracts. Additionally, to avoid hallucinations caused by excessively long inputs [29, 43, 59], *V2E* provides only essential information to the LLM, such as constructors, potential vulnerability paths, and vulnerability features. Then, *V2E* guides the LLM to reason step-by-step and output the PoC. We will give more details in Section 4.1.

Trigger and Profit-Driven PoC Effectiveness Analysis. Unlike vulnerability detection, using PoC to verify smart contract vulnerability must answer two questions: whether the vulnerability is triggerable and causes real harm. Existing methods are difficult to achieve this purpose. Specifically, static analysis [19, 26] or LLM-based approaches [34, 50] rely on code patterns to identify vulnerabilities, making them unable to handle complex state updates and operation stages in PoCs. Additionally, fuzzing methods [38, 61] cannot distinguish state changes caused by cheatcodes, limiting their ability to process intricate interactions in PoC execution. Meanwhile, relying on inline commands [17] neither provides fine-grained execution context nor offers extensibility.

As mentioned before, *V2E* verifies vulnerability through triggerability and profitability analysis of PoC. In detail, to avoid the influence of cheatcodes on the results, *V2E* executes the PoC in a customized EVM and collects the execution context to analyze whether the vulnerability has been triggered. Besides, *V2E* records the balance changes during PoC execution and feeds them into LLM for profitability analysis. The key insight here is that in smart contracts, assets can take various forms, including native tokens and other types of tokens, and may involve not only the attacker but also other users. By considering the type and source of each asset, the LLM can accurately determine whether the attacker has gained profit. More details are shown in Section 4.2

Feedback-Driven PoC Refinement. As mentioned earlier, PoCs generated by LLMs may be invalid, as LLMs cannot account for the impact of dynamic state changes on execution. Therefore, guiding the PoC to explore vulnerability is essential. Since LLM outputs lack diversity [63], directly asking LLM to regenerate a PoC is ineffective. Instead, updating the PoC based on execution feedback, such as control flow and data flow changes, is a feasible method and is commonly used in vulnerability detection [33, 38, 45]. However, LLM cannot directly utilize this feedback, as smart contracts can only provide bytecode-level feedback during execution.

V2E feeds the execution context and the failed PoC into LLM for refinement. Although the LLM's nondeterministic nature may affect how the PoC is modified, it will not affect the performance of *V2E*, as Step 2 in *V2E* validates the refined PoC and thus ensures its correctness. Specifically, there are two scenarios in which the PoC needs to be adjusted: (1) the PoC execution fails. In this case, *V2E* collects execution bytecode and identifies the most critical bytecode, like failed checks,

or incorrect state read/write. Next, *V2E* uses Source Mapping to recover the corresponding source code from the bytecode. Source Mapping [48] is a file generated by the compiler during compilation. It provides detailed records of which source code fragments correspond to each bytecode. Finally, the critical source code and PoC are fed into LLMs for updates; (2) the PoC executes successfully, but the exploitability of the vulnerability cannot be evaluated. Rather than directly modifying parameters, *V2E* improves PoC through primitive operations. Primitive operations are a set of predefined PoC update strategies designed based on vulnerability characteristics and stages of PoC execution. The insight here is that different vulnerabilities require different exploitation strategies. For example, in Transaction Order Dependence, the sequence of transactions has a greater impact, while in Randomness from Chain Attributes, finding suitable block properties is more critical. By guiding PoC updates based on the characteristics of the vulnerability, it becomes more likely to confirm the exploitability. More details are shown in Section 4.3.

PoC Synthesis Prompt Template

System: You are a smart contract expert, and you need to analyze the smart contract I provide and generate a valid PoC to verify the vulnerability. You must follow the user-defined rules. You can only output one runnable file of PoC without any explanations or markdown.

Contract Initialization: [Constructor] is the constructor of the contract. You need to analyze the purpose of each parameter and generate appropriate initialization values to deploy the contract.
Vulnerability Path: [Function] may have Vulnerability, and [Related Function] is the function that can modify the state that [Function] depends on. Try to combine these functions and the additional information I provide below to trigger the vulnerability and make profits.
Vulnerability Features: .....

PoC Requirements: You need to use forge-std/Test.sol and function setUp to initiate the environment, and exploit vulnerability in function testExploit.
Additional Information: e.g., constant states read by [Function], block number to be tested.....

[Function]
[Constructor]

Fig. 5. The prompt template used for PoC synthesis.

3.2 Workflow of *V2E*

Fig. 4 shows the detailed workflow of *V2E*. *V2E* takes contract, function under test, and vulnerability report as input, automatically verifies the vulnerability through PoC, and outputs exploited vulnerability. Specifically, in Vulnerability Path Guided PoC Synthesis, *V2E* uses fine-grained static analysis to identify potential vulnerability entry functions in the contract and constructs vulnerability paths based on these functions. Then, *V2E* incorporates contract initialization requirements, vulnerability paths, and vulnerability characteristics into the prompt, guiding the LLM to generate PoCs. After that, these PoCs are stored in PoC Corpus for further analysis. In Trigger and Profit-Driven PoC Effectiveness Analysis, *V2E* retrieves one PoC at a time from the corpus for execution and collects execution context to evaluate its validity. When *V2E* identifies a PoC that can both trigger vulnerability and make profits, it outputs the PoC and confirms the vulnerability. Otherwise, *V2E* employs PoC upgrade. In Feedback-Driven PoC Refinement, if execution fails, *V2E* extracts and analyzes critical bytecode. Simultaneously, if no evidence is found, *V2E* improves the PoC by using primitive operations and adds the updated PoC into the corpus for further analysis.

4 Approach Details

4.1 Vulnerability Path Guided PoC Synthesis

To generate specific PoCs, *V2E* employs fine-grained static analysis to analyze potential vulnerability paths and includes detailed deployment requirements along with vulnerability features in the prompt, guiding the LLM to reason step-by-step and produce PoCs.

Vulnerability path analysis. For a given function under test, *V2E* uses the following formula to find potential entry functions:

$$F_{dep}(f_{test}) = \{f \in C | Write(f) \cap Read(f_{test}) \neq \emptyset \& \neg Access(f)\} \tag{1}$$

where f_{test} represents the function under test, F_{dep} denotes the set of entry functions for f_{test}, C refers to the contract under test, $Write$ represents the set of states updated by function, $Read$ signifies the set of states read by the function, $Access$ is the access control mechanism in f. Specifically, this formula is composed of two components: state read-write dependency analysis and access control analysis. State read-write dependency analysis first identifies the states that the tested function depends on, and then searches the contract for entry functions that can modify those states. After that, access control analysis determines whether the function has access control by primarily analyzing the presence of modifiers like `onlyOwner`, as modifiers are widely used to implement access control in smart contracts (over 66% of contracts [12]). Moreover, although contracts may implement access control through other mechanisms (e.g., `require`), resulting in incorrect PoCs, the PoC effectiveness analysis in Section 4.2 can substantially mitigate such false positives.

PoC synthesis. For each f in $F_{dep}(f_{test})$, *V2E* attempts to combine f_{test} and f to construct a potential exploit path and generate the PoC. In detail, *V2E* employs the prompt template shown in Fig. 5. This template is composed of three sections. It begins with a system prompt that provides an overview of the task and sets the LLM's role. The middle part is the user prompt, which includes three steps for PoC generation. For subsequent vulnerability testing, *V2E* instructs the LLM to first deploy the contract based on constructor parameters. Then, it generates the PoC according to the vulnerability path and specific vulnerability characteristics. The final part of the prompt provides the constraints for PoC synthesis. Specifically, *V2E* utilizes Foundry for PoC execution and evaluation. To standardize the testing process, *V2E* mandates the use of `setUp` for test initialization and `testExploit` as the entry point for the exploit. Besides, to adapt different requirements for various vulnerabilities and contracts, *V2E* also injects additional information, like contract invariants or block number, and instructs the LLM on its proper utilization.

V2E stores all generated PoCs in the PoC Corpus and continuously retrieves PoCs from the corpus for execution, evaluating and updating them based on their triggerability and profitability.

4.2 Trigger and Profit-Driven PoC Effectiveness Analysis

V2E mainly performs PoC effectiveness analysis in this step. Particularly, *V2E* feeds the PoC into a customized EVM for execution. First, the EVM invokes function `setUp` in the PoC to initialize the execution environment, such as deploying the vulnerable contract and allocating sufficient funds. *V2E* ignores all actions performed within `setUp` to ensure the accuracy of the PoC evaluation. Then, *V2E* invokes function `testExploit` in the PoC to perform the attack and collects the execution context during this process. Note that in this step, *V2E* ignores all control-flow and data-flow changes caused by cheatcode operations. Finally, *V2E* checks step-by-step whether the PoC executes successfully, triggers the vulnerability, and generates profit. If any step yields a negative result, *V2E* terminates the analysis and proceeds to update the PoC with strategies in Section 4.3.

Execution analysis. After PoC execution, *V2E* analyzes whether the final opcode executed by the PoC is `revert`, as the EVM returns opcode `revert` when execution fails. Therefore, if it is `revert`, the PoC is marked as a failure. Otherwise, execution is deemed successful.

Table 1. Vulnerability trigger rules used in *V2E*.

Vulnerability	**Trigger Rules**
Unprotected Ether Withdrawal	$Transfer(tx.origin)$
Unprotected Selfdestruct	$op == \texttt{Selfdestruct}$
Reentrancy	$\exists c_j \in c_i, i \neq j \&\& func(c_i) == func(c_j)$
Transaction Order Dependence	$write(slot) \in c_i, read(slot) \in c_j \&\& j > i$
Randomness from Chain Attributes	$op \in \{\texttt{Timestamp}, \texttt{Blockhash}, \texttt{Number}\}$

Trigger analysis. To determine whether PoC triggers a vulnerability during execution, *V2E* defines different trigger rules for each vulnerability. The detailed rules are shown in Table 1.

- **Unprotected Ether Withdrawal**. *V2E* checks whether the PoC contains any $Transfer$ to the transaction sender $tx.origin$.
- **Unprotected Selfdestruct**. *V2E* determines whether the PoC invokes opcode $Selfdestruct$.
- **Reentrancy**. *V2E* analyzes whether two calls c_i and c_j occur in execution, where c_j is a nested call inside c_i and they invoke the same function.
- **Transaction Order Dependence**. *V2E* examines whether the PoC contains two calls, c_i and c_j, such that c_i updates a specific storage $slot$ and c_j subsequently reads $slot$'s value.
- **Randomness from Chain Attributes**. *V2E* identifies whether there is an opcode op related to reading block attributes, like block number, timestamp, and block hash.

When the corresponding vulnerability rule is observed during PoC execution, *V2E* considers the PoC to have successfully triggered the vulnerability.

Profit analysis. *V2E* evaluates the attacker's profit to assess the severity of the vulnerability. Specifically, *V2E* instructs the PoC to output the attacker's balance changes before and after the exploit via EVM events. Since blockchains contain native assets such as Ether, as well as various types of tokens, such as ERC-20 tokens and non-fungible tokens (NFTs), these balance changes are then fed into LLM to determine whether the exploit is financially profitable.

Algorithm 1 Failed PoC Update Algorithm.

Input: Executed bytecode list L_b, failed PoC PoC_{fail}, Contract source code s
Output: Updated PoC PoC_{update}

```
msg_revert := analyzeOutput(L_b)
for b in L_b.reverse() do
    sourceMapping := compile(s)
    sourceCode := sourceMapping.convert(b)
    if type(sourceCode) in {Check, State Update} then
        loc := source_code // code where execution failed
        func := searchFunction(loc) // function where execution failed
    end if
end for
PoC_update := failedPocUpdate(PoC_fail, loc, func, msg_revert)
```

Severity analysis. *V2E* terminates execution either when a valid PoC is found or when a predefined execution timeout is reached. Upon termination, *V2E* analyzes the execution results of all PoCs in the corpus and provides one of the following three conclusions:

1. **Exploitable**: If *V2E* identifies a PoC that both successfully triggers the vulnerability and yields profit for attackers, it outputs $Exploitable$, indicating that the vulnerability poses a real threat.

$$\exists PoC \in corpus, Trigger(PoC) \&\& Profit(PoC) => Exploitable \tag{2}$$

2. **Non-exploitable**: If, by the end of validation, *V2E* finds that there is a PoC that can either trigger the vulnerability without profit or generate profit without triggering the vulnerability, it considers the vulnerability to be of low severity or nonexistence, and outputs $Non-exploitable$.

$$\begin{aligned} &\exists PoC \in corpus, Trigger(PoC) \&\& \neg Profit(PoC) || \\ &\neg Trigger(PoC) \&\& Profit(PoC) => Non-exploitable \end{aligned} \tag{3}$$

3. **Manually Check**: If no PoC is found that can either trigger the vulnerability or generate profit, *V2E* outputs *Manually Check*, suggesting that the vulnerability requires further exploration.

$$\forall PoC \in corpus, \neg Trigger(PoC) \&\& \neg Profit(PoC) => Manually\ Check \tag{4}$$

4.3 Feedback-Driven PoC Refinement

When the PoC fails verification, *V2E* performs PoC optimization based on the execution feedback through bytecode semantic analysis and primitive operation.

Bytecode semantic-based PoC update. If a PoC fails, *V2E* uses the algorithm shown in Algorithm 1 to update the PoC. Specifically, *V2E* collects all the bytecode from the execution and analyzes it to extract the execution failure output. Then, *V2E* analyzes bytecode in reverse order to find the critical point of failure. To confirm the semantics of the bytecode, *V2E* first compiles the contract related to the bytecode and retrieves the source mapping file. *V2E* then locates the source code fragment corresponding to the bytecode through this file. When finding a bytecode whose related source code is a check (e.g., `require` or `assert`) or a state-update statement, *V2E* identifies this as the critical failure statement. *V2E* focuses on these two types of statements because contract execution failures are typically caused by failing checks or state-update errors, such as transferring with insufficient balance. Finally, *V2E* provides the failing statement, its corresponding function, and the failure reason output by the EVM to the LLM for the PoC update. The updated PoC is added to the corpus for further in-depth analysis.

Table 2. Primitive operation used in *V2E*. TA=Trigger analysis. PA=Profit analysis.

Name	Primitive Operation	Vulnerability					Stage	
		RE	RCA	TOD	UEW	US	TA	PA
add_user	Add additional users and let them invoke {function}.		■	■	■			■
change_invoker	Change the invoker of {contract}.	■			■	■	■	
change_order	Swap the order of attackers and other users.			■	■			■
modify_block	Modify the {block_attribute}.		■				■	
change_argument	Change the argument or value of {function}.	■	■	■	■	■	■	■

Primitive operation guide PoC improvement. A PoC may execute without errors but still fail to validate a vulnerability, for instance, by not confirming its triggerability or profitability. In such cases, *V2E* optimizes the PoC by selecting appropriate operations from a predefined set, based on the vulnerability type and the specific stage of failure. The specific names and rules of primitive operations are shown in Table 2.

- **add_user**. This operation is designed to enhance the profitability of RCA, TOD, and UEW. In detail, these vulnerabilities are all state-sensitive. However, after the PoC is deployed,

the contract is in initial state, and the state-modifying operations provided by PoC may be insufficient. Thus, *V2E* simulates state changes by adding additional user operations.

- **change_invoker**. This operation is used to aid in exploring RE, UEW, and US. These vulnerabilities may impose constraints on the caller. For example, the vulnerable path may only be triggered when invoked by the deployer. To address this, *V2E* attempts to explore different execution paths by varying the function caller.
- **change_order**. This operation is designed to improve the profitability of TOD and UEW, which are sensitive to the order of function calls. *V2E* explores profitability by performing execution reordering to trigger different sequences of state modifications.
- **modify_block**. This operation is used to explore the triggerability of RCA. Since RCA relies on block attributes, such as the block number and timestamp, to simulate randomness, *V2E* attempts to trigger different execution paths by modifying current block properties.
- **change_argument**. This operation applies to all types of vulnerabilities and all stages of analysis. By modifying function parameters and the amount of Ether sent with function calls, *V2E* attempts to trigger alternative execution paths and explore how different state conditions impact the profitability.

For all selected suitable primitive operations, *V2E* sequentially applies them to modify the PoC. All newly generated PoCs are then added back into the corpus for further analysis.

5 Evaluation

In this section, we first provide the datasets used in the evaluation and introduce our evaluation setup. Then, we show the evaluation results of *V2E*.

Table 3. The distribution of reported vulnerabilities in our dataset. We manually verify and classify each vulnerability based on whether it is indeed exploitable.

		UEW	US	RE	TOD	RCA	Sum
D_{manual}	**Exploitable**	8	1	29	4	8	50
	Non-Exploitable	7	0	2	1	4	14
$D_{onchain}$	**Exploitable**	0	1	6	45	22	74
	Non-Exploitable	1	1	55	31	38	126
Total		16	3	92	81	72	264

5.1 Implementation and Evaluation Setup

Dataset. We collect 264 vulnerable contracts from SmartBugs [14], as it provides a large number of open-source and peer-reviewed contracts. These contracts are divided into D_{manual} and $D_{onchain}$ based on their origin. Specifically, D_{manual} contains 64 manually constructed vulnerable smart contracts, while $D_{onchain}$ consists of 200 real-world contracts deployed on-chain. Note that SmartBugs contains vulnerabilities that are out of our scope, such as Unchecked Low-Level Calls, as well as some contracts that are incompatible with Foundry [15]. Therefore, we apply the following procedures to construct both datasets. The distribution of vulnerabilities is shown in Table 3.

• **Filtering vulnerabilities.** As mentioned earlier, *V2E* refers to the common SWC categories summarized by real-world audits [69] and selects five types of vulnerabilities based on their potential for profit. Since D_{manual} does not provide SWC labels, we manually craft PoCs, collect execution traces (i.e., paths) and fine-grained state changes during PoC execution. Then, we leverage the SWC features to select 64 vulnerable contracts. For $D_{onchain}$, we perform filtering based on the SWC tags included in the dataset. Initially, $D_{onchain}$ contains 761 vulnerable contracts. Due to time constraints, we choose a sample size of 200 based on a 90% confidence level and a 5% margin of error.

We then determine the number of vulnerabilities in each category according to their distribution and conduct sampling accordingly.

• **Filtering out contracts with complex constructor parameters.** When contract deployment depends on complex parameters, such as addresses or bytes, *V2E* may not know what values to assign, leading to test failures. Therefore, contracts with such deployment requirements are excluded from evaluation. However, it is worth noting that this procedure does not affect the representativeness of the dataset. Specifically, we inspect 31 contracts excluded during sampling and find that their vulnerability patterns are fully covered by remaining cases, and 87% of complex parameters are irrelevant to vulnerable locations.

• **Updating contract version to above 0.6.** To satisfy the minimum Solidity version supported by Foundry (i.e., 0.6) [16], we manually upgrade all contracts written in versions earlier than 0.6. Note that this upgrade is restricted to syntax-level compatibility adaptations required for compilation, such as updating deprecated keywords or language constructs. The whole process does not involve any intentional modification of the contract logic, control flow, or behavior. To validate that these changes do not alter contract semantics, we construct a PoC for each contract that requires an upgrade. Then, we check whether the PoC maintains consistent results before and after update. The results show that no vulnerabilities exhibit different behaviors due to the update.

• **Labeling the severity of vulnerability.** SmartBugs only provides vulnerability labels but does not include information on the exploitability, like whether they are triggerable and profitable. Therefore, we invite two experts with at least two years of experience in smart contract auditing to assess 264 vulnerabilities independently. Specifically, if an expert determines that a vulnerability could be exploited, they are required to write a PoC. When the two experts reach a consensus, we label exploitable vulnerabilities as Exploitable and others as Non-Exploitable. In cases where the two experts disagreed, a third expert is brought in to validate the PoC and provide a final decision.

Implementation. *V2E* leverages the Foundry-based execution engine provided by ityFuzz and further builds customized PoC execution and fine-grained state collection logic on top of it, enabling PoC validity analysis. Besides, *V2E* implements the PoC synthesis and refinement modules from scratch. All experiments in our evaluation are conducted on a machine with two Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz, 512GB RAM, and Ubuntu 20.04.4 OS. During the experiments, we use three models: GPT-4o (2024-11-20) [40], DeepSeek R1 (2025-05-28) [22], and Claude Sonnet 4 (2025-05-14) [1]. By default, *V2E* primarily uses GPT-4o in the experiments. Regarding model parameters, for PoC generation and refinement, we set the temperature to 1 to encourage the model to produce more diverse PoCs. In all other stages, we set the temperature to 0 to improve determinism and stability. For the remaining parameters, we follow the configuration used in GPTScan [50]. In our preliminary experiments, we observe that over 70% of vulnerabilities can be confirmed within 10 minutes. Moreover, longer execution times increase LLM token overhead without providing benefits. Thus, we set a time limit of 30 minutes for trade-offs. When the timeout is reached, *V2E* terminates the analysis and produces a final conclusion based on the results of all PoCs. Besides, we find that providing prior context biases the model toward producing responses that are similar to earlier outputs, regardless of whether those previous responses are correct. Therefore, *V2E* clears context before each LLM request.

Evaluation metrics. In this paper, we use TP (True Positive) to denote correctly identifying an exploitable vulnerability, FP (False Positive) to denote incorrectly labeling a non-exploitable vulnerability or contract without vulnerability as exploitable, TN (True Negative) to denote correctly identifying a non-exploitable vulnerability or contract without vulnerability, and FN (False Negative) to denote misclassifying an exploitable vulnerability as non-exploitable.

Research questions. We primarily focus on the following four research questions:

RQ1. How effective is *V2E* in validating exploitable vulnerability?
RQ2. How effective is *V2E* compared with existing methods?
RQ3. How effective is *V2E* in eliminating false alarms?
RQ4. What is the impact of each module in *V2E* on verifying vulnerabilities?
RQ5. What is the performance overhead of *V2E*?

Table 4. The verification results of *V2E*. MC (Exploit | Non-Exploit) refers to instances that require further manual check (inspection).

	D_{manual}					$D_{onchain}$				
	TP	**TN**	**FP**	**FN**	**MC**	**TP**	**TN**	**FP**	**FN**	**MC**
UEW	6	5	1	0	2 | 1	0	1	0	0	0 | 0
US	1	0	0	0	0 | 0	1	1	0	0	0 | 0
RE	28	1	0	1	0 | 1	6	28	2	0	0 | 25
TOD	3	1	0	0	1 | 0	41	8	4	1	3 | 19
RCA	6	3	0	0	2 | 1	10	23	2	5	7 | 13
Total	44	10	1	1	5 | 3	58	61	8	6	10 | 57

5.2 Effectiveness of *V2E* in Validating Exploitable Vulnerability

To answer RQ1, we evaluate *V2E* with D_{manual} and $D_{onchain}$. Specifically, for each PoC that *V2E* deems exploitable, we let two experts independently audit it. If both experts agree that the PoC successfully triggers the vulnerability and enables the attacker to profit, we consider it a true positive identified by *V2E*. In cases of disagreement, a third expert is invited to review the PoC and provide the final judgment. Table 4 shows the final results. As shown in the table, *V2E* successfully confirms 102 out of 124 exploitable vulnerabilities, achieving a precision of 91.9% and a recall of 82.3%. Meanwhile, it correctly identified 71 out of 140 non-exploitable vulnerabilities, accounting for 50.7%. Note that 4 FPs, 1 FN, and 3 exploitable MCs and 12 non-exploitable MCs are caused by LLM analysis error. We will provide a further analysis of these cases in Section 5.5.

False Positives. Through the analysis of false positives, we identify the following main reasons: (1) Lack of specific token prices (2 FPs). Since token prices are determined by the markets and change over time, *V2E* does not know the price of tokens (e.g., ERC20 tokens) in the contract and defaults to a 1:1 exchange rate with native tokens, like Ether. For example, if the attacker spends 1 ether to obtain 100 tokens, *V2E* considers the attacker to have made a profit of 99 tokens. However, this approach may lead to incorrect estimations of potential attack profits, resulting in false positives. (2) Bypassing checks that should not be bypassed (3 FPs). When execution fails due to permission checks, such as requiring the caller to be the contract deployer, *V2E* may attempt to directly let the attacker deploy the contract and bypass the check.

```
1 bytes32 hash = 0x12312ab.....;
2 function solve (string memory ) public payable {
3   if(hash==keccak256(memory)) {
4     msg.sender.call{1000 ether};
5 }}
                    hard to find through random generation
```

Fig. 6. A False Negative caused by a lack of mathematical reasoning capabilities.

False Negatives and Manually Check. The causes of false negatives and exploitable Manually Check are similar, and can be attributed to the following reasons: (1) *V2E* fails to bypass `tx.origin` (2 FNs and 5 exploitable MCs). All transactions in the PoC are executed as internal transactions, meaning the caller is a fixed externally owned account. However, some contracts enforce `tx.origin`

checks to ensure that only authorized users can trigger certain functions. As a result, the PoC repeatedly fails the check and cannot proceed with the exploit. (2) *V2E* lacks complex mathematical capabilities (4 FNs and 7 exploitable MCs). Due to the limited arithmetic reasoning ability of LLMs, *V2E* struggles to infer precise inputs when the contract involves complex computations. For example, as illustrated in Fig. 6, there exists a profitable TOD in function `solve`. In a real-world scenario, once a user discovers a valid input and sends a transaction, the attacker can frontrun the transaction by replicating the input. Although *V2E* successfully generates the correct transaction sequence structure, it can only attempt to trigger the vulnerability by randomly generating candidate inputs, lacking the ability to compute them directly.

For non-exploitable Manually Check, the primary reason lies in *V2E*'s conservative strategy (48 non-exploitable MCs). Specifically, to reduce the impact of LLM uncertainty, *V2E* only proceeds to assess vulnerability when the PoC passes either the triggerability or profitability analysis. However, some vulnerabilities are merely theoretical and fail during actual execution. In such cases, *V2E* cannot perform trigger and profit analysis, and ultimately leads to Manually Check.

Table 5. The validation consequences compared *V2E* with $A1_{re}$ and LLM_{multi}.

	D_{manual}			$D_{onchain}$			Total					
	TP	**FP**	**FN**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**	**Pre.**	**Rec.**	**F1**
$\boldsymbol{A1_{re}}$	34	8	61	29	51	45	63	59	61	51.6%	50.8%	51.2%
$\boldsymbol{LLM_{multi}}$	31	6	14	12	46	60	43	52	74	45.2%	34.7%	39.3%
V2E	44	1	1	58	8	6	102	9	7	91.9%	82.3%	86.8%

5.3 Comparison with Existing Methods

Comparison with A1. As mentioned earlier, the most relevant tool to *V2E* is A1, a SOTA framework generating executable PoCs for smart contract vulnerability detection [17]. However, since the original A1 only provides a demo version [4] to the public, we cannot perform an apples-to-apples comparison. As our best effort, based on the framework in [4], we reimplement A1 step by step, following the procedure and its core prompts described in the A1 paper. We refer to this re-implementation version as $A1_{re}$. To ensure the reliability of $A1_{re}$, we evaluate it using the original A1's benchmark (36 incidents). Note that, as our best effort, we follow the same parameters specified in A1. For example, we aggregate the outputs of the same six models to derive the final answer, running two trials with five rounds per iteration. In the end, $A1_{re}$ identifies 22 incidents, whereas A1 identifies 26. These results indicate that $A1_{re}$ is relatively comparable to A1. Finally, we apply $A1_{re}$ to D_{manual} and $D_{onchain}$, and compare the results with *V2E*. As shown in Table 5, *V2E* significantly performs better than $A1_{re}$. Particularly, $A1_{re}$ incurs much higher FPs and FNs than *V2E* on the two datasets. The high efficacy of *V2E* is mainly attributed to the adoption of off-chain testing and our customized PoC executing environment. In this way, *V2E* can collect fine-grained context information and explore a larger state space, thereby validating more exploitable vulnerabilities.

Comparison with multi-agent and static-based methods. To better demonstrate the effectiveness of *V2E*, we draw inspiration from iAudit [34] and use a framework combined with multi-agent and static analysis, called LLM_{multi}, to analyze vulnerabilities in both D_{manual} and $D_{onchain}$. Specifically, we first generate PoCs based on vulnerability path guided PoC synthesis for the given vulnerability. Then, an LLM analyzes whether the PoC successfully triggers the vulnerability and yields profit, providing a detailed justification. Finally, another LLM evaluates the reasoning. If the justification is deemed valid, a conclusion is produced. Otherwise, the process loops back to reassess and regenerate the explanation. The detailed results are shown in Table 5. In detail, LLMs

are unable to execute code and obtain instantaneous runtime state values. As a result, they assess vulnerability exploitability solely based on static code snippets. This limitation leads to inaccurate judgments and introduces a large number of false positives and false negatives.

Table 6. The results of using *V2E* to eliminate false alarms in Slither, Mythril, and Confuzzius.

	Original Tool (Baseline)			Verification with *V2E*				
	TP	**FP**	**FP Rate**	**TP**	**FP**	**FN**	**FP Rate**	**FP Elimination**
Slither	37	192	83.8%	27	2	4	6.9%	-76.9%
Mythril	39	78	66.6%	28	3	2	9.7%	-56.9%
Confuzzius	46	106	69.7%	41	2	2	4.7%	-65%

5.4 Effectiveness of *V2E* in Eliminating False Alarms

To answer RQ3, we use two actively maintained static analysis tools, Slither [13] and Mythril [37], as well as one fuzzing tool, Confuzzius [51] to identify vulnerabilities in both D_{manual} and $D_{onchain}$. We invite two experts to independently validate the detection results produced by the three tools. If they agree that a reported issue is a true vulnerability, they write a PoC. If there is any disagreement between the two experts, a third expert will assess the PoC validity, and the three experts together will reach a final decision. Note that we do not perform a comprehensive recall analysis by aggregating the results of the three tools, as *V2E* mainly aims to validate whether reported vulnerabilities are exploitable, rather than detecting vulnerabilities. In practice, if a detection tool misses a vulnerability, *V2E* cannot conduct further analysis. In addition, although *V2E* may misclassify real vulnerabilities as FNs, this rarely occurs (with recall of 82.3% in our experiments). Besides, we exclude other testing tools such as ityFuzz [45], Verite [27], or Sailfish [3], as they do not cover all vulnerabilities *V2E* targets.

We then use *V2E* to analyze the detection results reported by these tools. We let two experts review the outcomes of *V2E*. If *V2E*'s detection result is inconsistent with the ground truth, we label it as either a false negative or a false positive. As shown in Table 6, although the detection tools report a large number of vulnerability warnings, most of them are false positives. In contrast, by analyzing both triggerability and profitability, *V2E* successfully assesses the majority of these vulnerabilities, effectively reducing false alarms.

5.5 Effectiveness of Each Module in *V2E*

To answer RQ4, we use D_{manual} and $D_{onchain}$ to analyze the impact of different LLMs and individual components on *V2E* overall effectiveness.

Table 7. The vulnerability severity verification results under different LLMs.

	TP	**FP**	**FN**	**Pre.**	**Rec.**	**F1**
GPT-4o	102	9	7	91.9%	82.3%	86.8%
DeepSeek R1	74	7	31	91.4%	59.7%	72.2%
Claude Sonnet 4	95	11	19	89.6%	76.6%	82.6%

Impact of different LLMs. We evaluate the performance of *V2E* under GPT-4o, DeepSeek R1, and Claude Sonnet 4, respectively. The results are shown in Table 7. Since PoC generation and update involve understanding the code and the vulnerability, and generating new code, SOTA LLMs perform differently. Specifically, we find that GPT-4o identifies the highest number of true positives and has the fewest false negatives. DeepSeek R1 produces the lowest number of false positives, while Claude Sonnet 4 results in the fewest cases requiring manual confirmation. Overall, GPT-4o achieves the best precision and recall among the three models. Through the analysis, we

find that the primary cause of LLM misjudgments is the lack of sufficient source code context. This is because *V2E* explores a single entry function for each PoC. However, when a vulnerability requires manipulation of multiple functions, the PoC's effectiveness is compromised because LLM lacks context to invoke. This design is a deliberate trade-off, as PoC synthesis is a complex task, and minimizing the LLM's reasoning load is crucial for improving generation outcomes.

Table 8. Detection results of *V2E*, without vulnerability path analysis, without exploit and profit-driven analysis, and without feedback-driven PoC fine-tuning based on D_{manual} and $D_{onchain}$.

	TP	TN	FP	FN	MC
Without vulnerability path analysis	29	46	10	45	134
Without trigger rule-based analysis	44	59	91	65	5
Without LLM-based profit analysis	62	63	32	79	28
Without feedback-driven PoC refinement	37	58	9	45	115
V2E	102	71	9	7	75

Effectiveness of vulnerability path analysis. To verify the impact of vulnerability path analysis on PoC generation, we remove it. Instead, we provide the LLM with all function signatures in the contract and instruct it to generate an exploit path based on its understanding of the vulnerability. The result is shown in Table 8. Although the vulnerability data in SmartBugs has been used to train LLMs, it is still difficult for LLMs to generate effective PoC. This highlights the necessity of providing LLMs with sufficient information (i.e., vulnerability path and potentially vulnerable functions). Moreover, excessively long inputs increase the likelihood of hallucinations. As a result, it fails to generate valid PoCs in many cases, leading to many vulnerabilities remaining unconfirmed.

Effectiveness of trigger rule-based analysis. We separately investigate the effectiveness of trigger and profit analysis. To validate the trigger rules, we ask the LLM to directly judge the PoC. Specifically, if LLM considers the PoC valid, the PoC is marked as triggerable. Otherwise, we select corresponding primitive operations to update PoC based on the reason and the type of vulnerability. The results are shown in Table 8. Since LLM does not execute code, it tends to overlook how runtime states affect vulnerability triggerability, resulting in more FPs and FNs.

Effectiveness of LLM-based profit analysis. To validate the effectiveness of profitability analysis, we design a set of rules to determine whether an attack yields profit. Specifically, we collect all native-token transfers (e.g., ETH) and token transfer logs (e.g., ERC-20 Transfer events) produced during execution to analyze the attacker's profit. As shown in Table 8, due to the heterogeneity of assets in smart contracts, rule-based profit checking may miss certain value-transfer signals, leading to more false negatives.

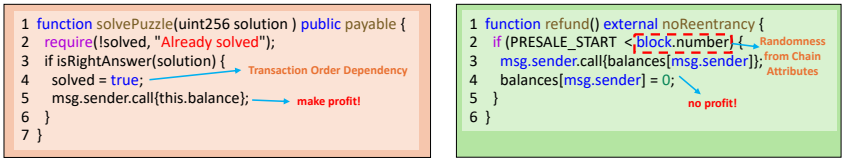
Effectiveness of feedback-driven PoC refinement. To evaluate the effectiveness of feedback-guided PoC exploration, we remove this component from the pipeline. Specifically, when a PoC requires updating, *V2E* directly feeds the original PoC and its execution results into the LLM and prompts it to generate a new PoC. As shown in Table 8, due to the lack of fine-grained execution feedback, LLM cannot identify the specific reasons why a PoC is invalid. Moreover, consistent with observations in prior studies [11, 58], we find that, without additional constraints, LLMs tend to make only local edits, such as changing parameter values, rather than revising the code structure, regardless of the temperature setting. As a result, the generated PoCs are overly similar, which limits their ability to explore vulnerability.

5.6 Overhead of *V2E*

V2E takes an average of 17.5 minutes to analyze a vulnerability. The most time-consuming phase is PoC refinement, averaging 9.1 minutes. This indicates that PoCs generated by LLM are often

not immediately suitable for validating vulnerability and require continuous iteration by *V2E*. The second-largest time cost is PoC compilation, with an average cost of 6.3 minutes. This is because *V2E* needs to compile both the initially generated and all subsequently updated PoCs during the validation process. In contrast, since *V2E* is an off-chain testing tool, PoC execution is very fast, taking an average of 1.8 minutes. Therefore, the time overhead of *V2E* is acceptable.

We record the token usage for each API request, including both input and output, and calculate the total cost based on OpenAI's official pricing [41]. The total cost of running *V2E* is 187.4 USD, averaging 0.71 USD per vulnerability. Consistent with the time cost analysis, PoC update incurs the highest token usage, with an average cost of 0.53 USD. Since *V2E* is able to provide only essential information to the LLM, the overall cost remains within an acceptable range.



(a) Harmful Transaction Order Dependence (b) Harmless Randomness from Chain Attributes

Fig. 7. Case study. Two real on-chain contract code snippets, each flagged as vulnerable by detection tools.

5.7 Case Study

In Fig. 7(a), function `solvePuzzle` contains a Transaction Order Dependence. Specifically, when a user discovers the correct `solution` and attempts to claim the reward via a transaction, an attacker can front-run the user by sending the transaction with identical parameters. During PoC generation, the LLM tends to place the attacker's actions after the user's, causing the PoC to fail *V2E*'s profitability check. In this case, *V2E* applies the primitive operation `change_order` to guide the LLM in swapping the order of the attacker and user actions. This successfully triggers a profitable exploit and *V2E* confirms the vulnerability's severity. In Fig. 7(b), line 2 relies on a block attribute and is thus categorized as having a Randomness from Chain Attributes. However, *V2E* observes that the generated PoC consistently fails to produce profit, and concludes that this is a false alarm, effectively filtering it out.

6 Discussion

6.1 Discussion

Philosophy of Manually Check. *V2E* uses Manually Check to enhance the effectiveness. Specifically, rather than forcing a definitive judgment on every vulnerability, Manually Check can enhance the interpretability and credibility of *V2E*. Besides, Manually Check category enables *V2E* to clearly define the boundary between vulnerabilities and false alarms, allowing for fine-grained classification and better vulnerability management for developers and auditors.

Suboptimal attack practices. The PoCs generated by *V2E* may not represent the optimal attack paths. For example, they might include redundant user actions or fail to yield the maximum possible profit. However, the primary goal of *V2E* is to determine whether a vulnerability is triggerable and profitable, rather than to identify the most efficient exploitation strategy. Moreover, since contract states can evolve over time, it is inherently difficult to assert how much profit can be obtained.

Testing with cheatcodes. *V2E* uses cheatcodes for pre-attack setup. For example, using `deal` to provide sufficient funds to the attacker and the contract, rather than acquiring capital through real-world mechanisms like flash loans. This approach simplifies the PoC workflow and increases

the likelihood of successful execution. Moreover, using cheatcodes to simulate realistic attack conditions is a common and accepted practice in smart contract vulnerability PoC generation [8].

6.2 Threats to Validity

Internal validity. Since *V2E* relies on contract source code for analysis and PoC generation, it is unable to handle contracts without available source code. However, this limitation is mitigated by two factors. First, most users tend to adopt open-source contracts and DApps, and a large portion of them are publicly available [67]. Second, *V2E* is designed for developers and auditors, who typically have access to the source code. Additionally, *V2E* currently lacks robust support for contracts that require complex constructor parameters or involve multi-token interactions, which may lead to inaccurate vulnerability assessments. As future work, we will explore how to leverage already-deployed contracts within a DApp to infer complex parameters and token exchange rate. Moreover, the limited mathematical capabilities of LLMs hinder *V2E*'s ability to handle complex arithmetic, leading to false negatives. In future work, drawing inspiration from agent-based approaches, we plan to explore how to integrate external tools to enhance *V2E*'s ability. Furthermore, although we enhanced the performance of *V2E* through program analysis and carefully crafted prompts, different models still yield varying performance across vulnerability types. In practice, aggregating results from multiple models can further enhance overall effectiveness.

External validity. Currently, *V2E* supports vulnerability validation only for Solidity [7] and Vyper [52] smart contracts. Specifically, this is because Foundry, the underlying execution framework, only supports these two languages. While other smart contract languages exist, such as Move [2], we believe the impact of this limitation is minimal. In detail, Solidity and Vyper are the first and fourth most widely used smart contract languages, respectively, and together manage over 88% of the total assets [9]. Besides, SmartBugs contains only a limited number of vulnerabilities, and its data has been used in LLM training, which could bias the results. However, on the one hand, sample size does not affect the effectiveness of *V2E*, as profit-based severity assessment is general and *V2E* derives representative characteristics from existing detectors and attack patterns. On the other hand, consistent with prior study [17], our experiments indicate that even when the information is provided, current models still struggle to generate valid PoCs. In future work, we will explore vulnerability injection [18] to enable a more comprehensive and fair evaluation. Additionally, due to the performance bottleneck caused by manually constructing the triggering rules, *V2E* currently covers only the five most common and financially exploitable vulnerabilities. Extending it to additional vulnerabilities requires extra engineering effort. In future work, we will explore automating the construction process. However, this limitation does not prevent *V2E* from adapting to new vulnerabilities. As a vulnerability validation tool, *V2E* can feasibly and inexpensively derive new patterns from existing exploits and detection tools.

7 Related Work

Smart contract vulnerability detection. A variety of methods have been proposed for smart contract vulnerability detection. For example, fuzzing generates inputs based on the contract's ABI and observes execution behavior and state changes to identify vulnerabilities. Tools such as Harvey [57] and sfuzz [38] apply greybox fuzzing for vulnerability detection, while IcyChecker [62], InsFinder [64], and SmartReco [65] leverage rich on-chain states to enable more efficient discovery of vulnerabilities. In contrast, static analysis detects vulnerabilities by examining contract source code or bytecode. For instance, DeFiTainter [26] performs static taint analysis at the bytecode level to detect price manipulation vulnerabilities, and Sailfish [3] uses static analysis on source code to identify state inconsistency issues. With the rise of LLMs, many studies have integrated LLMs

into vulnerability detection. For example, GPTScan [50] combines LLMs with static analysis to detect logical vulnerabilities in smart contracts, while PropertyGPT [32] applies LLMs to assist in formal verification of smart contracts. However, most of these detection approaches focus solely on identifying the presence of vulnerabilities, while neglecting their exploitability. As a result, users still need to spend significant effort analyzing the impact of the reported vulnerabilities.

Smart contract automatic exploit generation. Automatic exploit generation aims to demonstrate the existence of vulnerabilities by triggering them. One common approach is to use symbolic execution to explore potential vulnerability paths. For example, teEther [28] performs symbolic execution at the bytecode level to discover exploit paths, while FlashSyn [6] combines symbolic execution with counterexample synthesis to construct flash loan attack paths. Besides, ExGen [25] generates symbolic attack contracts and verifies their feasibility via symbolic execution. However, symbolic execution suffers from the path-explosion problem, which limits its applicability. Another widely used technique is fuzzing. ETHPLOIT [66] applies fuzzing to search for executable paths that can trigger vulnerabilities. AdvScanner [56] leverages LLM to generate and execute attack contracts targeting Reentrancy. However, these existing efforts primarily focus on detecting whether a vulnerability can be triggered, leading to lots of false alarms. In addition, as LLMs become increasingly capable of code generation and reasoning, some work leverages LLMs to generate PoCs and explore vulnerabilities from multiple perspectives. For example, XPLOGEN [10] feeds the LLM with known vulnerable contracts and exploits, and further incorporates DCR graphs to generate new vulnerable contracts and corresponding exploits. Besides, A1 [17] generates verifiable PoCs for vulnerability detection based on on-chain states and inline commands. In contrast, *V2E* uses a customized PoC execution environment to perform off-chain PoC generation and validation from scratch. Moreover, with the support of triggerability and profitability analysis, as well as feedback-driven refinement, *V2E* can effectively distinguish whether a reported vulnerability is truly exploitable, which is substantially more challenging.

8 Conclusion

In this paper, we present *V2E*, a specialized tool for validating exploitable smart contract vulnerabilities. Specifically, *V2E* synthesizes PoCs guided by vulnerability paths and verifies their validity based on triggerability and profitability analysis. Furthermore, *V2E* introduces a feedback-driven exploration method to increase the possibility of validating vulnerability. Evaluation demonstrates that *V2E* is effective in identifying exploitable vulnerabilities and eliminating false alarms. Moreover, it can help state-of-the-art tools in reducing false positives. Besides, *V2E* demonstrates strong performance in terms of both time efficiency and cost effectiveness.

9 Data Availability

To promote smart contract security development, we release the *V2E*. These resources can be accessed at https://github.com/jwzhang-zzz/V2E.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 62572497, No. 624B2139), NSFC-RGC Collaborative Research (No. 62461160332), Guangdong Zhujiang Talent Program (No. 2023QN10X561), and the Major Key Project of Pengcheng Laboratory under Grant PCL2025A07.

References

[1] anthropic. 2025. Introducing Claude 4. https://www.anthropic.com/news/claude-4.

[2] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. 2019. Move: A language with programmable resources. *Libra Assoc* 1 (2019).

[3] 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)*. IEEE, 161–178.

[4] c5huracan. 2025. Experimental exploration of AI-driven smart contract security analysis for educational and defensive research purposes. https://github.com/c5huracan/a1-agent-exploration.

[5] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. 2024. Smart contract and defi security tools: Do they meet the needs of practitioners?. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[6] Zhiyang Chen, Sidi Mohamed Beillahi, and Fan Long. 2024. FlashSyn: Flash Loan Attack Synthesis via Counter Example Driven Approximation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024*.

[7] Chris Dannen. 2017. *Introducing Ethereum and solidity*. Vol. 1. Springer.

[8] DeFiHackLabs. 2025. DeFi Hacks Reproduce - Foundry. Retrieved August 31, 2025 from https://github.com/SunWeb3Sec/DeFiHackLabs

[9] DefiLlama. 2025. Breakdown by Smart Contract Languages. Retrieved August 31, 2025 from https://defillama.com/languages

[10] Mojtaba Eshghie and Cyrille Artho. 2024. Oracle-guided vulnerability diversity and exploit synthesis of smart contracts using llms. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 2240–2248.

[11] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.

[12] Yuzhou Fang, Daoyuan Wu, Xiao Yi, Shuai Wang, Yufan Chen, Mengjie Chen, Yang Liu, and Lingxiao Jiang. 2023. Beyond "protected" and "private": An empirical security analysis of custom function modifiers in smart contracts. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1157–1168.

[13] 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.

[14] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2021. SmartBugs: a framework to analyze solidity smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1349–1352.

[15] Foundry-rs. 2025. A blazing fast, portable and modular toolkit for Ethereum application. Retrieved September 10, 2025 from https://github.com/foundry-rs/foundry

[16] Foundry-rs. 2025. Foundry support Solidity versions greater than 0.6. Retrieved August 31, 2025 from https://github.com/foundry-rs/forge-std/blob/master/src/StdToml.sol#L2

[17] Arthur Gervais and Liyi Zhou. 2025. Ai agent smart contract exploit generation. *arXiv preprint arXiv:2507.05558* (2025).

[18] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 415–427.

[19] 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.

[20] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. *Proc. ACM ICSE* (2023).

[21] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1176–1186.

[22] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[23] Tianyuan Hu, Jingyue Li, Bixin Li, and Andre Storhaug. 2024. Why smart contracts reported as vulnerable were not exploited? *IEEE Transactions on Dependable and Secure Computing* (2024).

[24] Jie Huang and Kevin Chen-Chuan Chang. 2023. Towards Reasoning in Large Language Models: A Survey. In *The 61st Annual Meeting Of The Association For Computational Linguistics*.

[25] Ling Jin, Yinzhi Cao, Yan Chen, Di Zhang, and Simone Campanoni. 2023. ExGen: Cross-platform, Automated Exploit Generation for Smart Contract Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* (2023).

[26] Queping Kong, Jiachi Chen, Yanlin Wang, Zigui Jiang, and Zibin Zheng. 2023. DeFiTainter: Detecting Price Manipulation Vulnerabilities in DeFi Protocols. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing*

and Analysis. 1144–1156.
[27] Ziqiao Kong, Cen Zhang, Maoyi Xie, Ming Hu, Yue Xue, Ye Liu, Haijun Wang, and Yang Liu. 2025. Smart Contract Fuzzing Towards Profitable Vulnerabilities. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 153–175.
[28] 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.
[29] Junyi Li, Jie Chen, Ruiyang Ren, Xiaoxue Cheng, Wayne Xin Zhao, Jian-Yun Nie, and Ji-Rong Wen. 2024. The dawn after the dark: An empirical study on factuality hallucination in large language models. *arXiv preprint arXiv:2401.03205* (2024).
[30] Wei Li, Yuhong Nan, Mingxi Ye, Jingwen Zhang, Peilin Zheng, and Zibin Zheng. 2025. ASTRO: Detecting Access Control Vulnerabilities in Smart Contracts via Graph Similarity Comparison. *IEEE Transactions on Software Engineering* (2025).
[31] 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*. 752–764.
[32] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2024. PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation. *arXiv preprint arXiv:2405.02580* (2024).
[33] Zhenguang Liu, Peng Qian, Jiaxu Yang, Lingfeng Liu, Xiaojun Xu, Qinming He, and Xiaosong Zhang. 2023. Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting. *IEEE Transactions on Information Forensics and Security* 18 (2023), 1237–1251.
[34] Wei Ma, Daoyuan Wu, Yuqiang Sun, Tianwen Wang, Shangqing Liu, Jian Zhang, Yue Xue, and Yang Liu. 2024. Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. *arXiv preprint arXiv:2403.16073* (2024).
[35] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. 2019. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. *Journal of Cases on Information Technology (JCIT)* 21, 1 (2019), 19–32.
[36] William Metcalfe et al. 2020. Ethereum, smart contracts, DApps. *Blockchain and Crypt Currency* 77 (2020), 77–93.
[37] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam* 9, 54 (2018), 4–17.
[38] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.
[39] Vikram Nitin, Baishakhi Ray, and Roshanak Zilouchian Moghaddam. 2025. FaultLine: Automated Proof-of-Vulnerability Generation Using LLM Agents. *arXiv preprint arXiv:2507.15241* (2025).
[40] OpenAI. 2025. OpenAI Platform. Retrieved August 31, 2025 from https://platform.openai.com/docs/models/gpt-4o
[41] OpenAI. 2025. OpenAI Price. Retrieved August 31, 2025 from https://openai.com/api/pricing/
[42] Daniel Perez and Benjamin Livshits. 2021. Smart contract vulnerabilities: Vulnerable does not imply exploited. In *30th USENIX Security Symposium (USENIX Security 21)*. 1325–1341.
[43] Han Qiu, Jiaxing Huang, Peng Gao, Qin Qi, Xiaoqin Zhang, Ling Shao, and Shijian Lu. 2024. LongHalQA: Long-Context Hallucination Evaluation for MultiModal Large Language Models. *arXiv preprint arXiv:2410.09962* (2024).
[44] Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. 2024. Llm4fuzz: Guided fuzzing of smart contracts with large language models. *arXiv preprint arXiv:2401.11108* (2024).
[45] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. Ityfuzz: Snapshot-based fuzzer for smart contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 322–333.
[46] Deniz Simsek, Aryaz Eghbali, and Michael Pradel. 2025. PoCGen: Generating Proof-of-Concept Exploits for Vulnerabilities in Npm Packages. *arXiv preprint arXiv:2506.04962* (2025).
[47] SmartContractSecurity. 2025. Smart Contract Weakness Classification and Test Cases. Retrieved August 31, 2025 from https://github.com/SmartContractSecurity/SWC-registry
[48] Solidity. 2025. Solidity Source Mappings - Documentation. Retrieved August 31, 2025 from https://docs.soliditylang.org/en/latest/internals/source_mappings.html
[49] soliditylang. 2025. Solidity v0.8.0 Breaking Changes. Retrieved September 10, 2025 from https://docs.soliditylang.org/en/latest/080-breaking-changes.html
[50] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[51] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE,

103–119.
[52] Vyper. 2025. Vyper Document. Retrieved August 31, 2025 from https://docs.vyperlang.org/en/latest/
[53] Zexu Wang, Jiachi Chen, Peilin Zheng, Yu Zhang, Weizhe Zhang, and Zibin Zheng. 2024. Unity is strength: Enhancing precision in reentrancy vulnerability detection of smart contract analysis tools. *IEEE Transactions on Software Engineering* (2024).
[54] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
[55] Shuohan Wu, Zihao Li, Luyi Yan, Weimin Chen, Muhui Jiang, Chenxu Wang, Xiapu Luo, and Hao Zhou. 2024. Are we there yet? unraveling the state-of-the-art smart contract fuzzers. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[56] Yin Wu, Xiaofei Xie, Chenyang Peng, Dijun Liu, Hao Wu, Ming Fan, Ting Liu, and Haijun Wang. 2024. AdvSCanner: Generating Adversarial Smart Contracts to Exploit Reentrancy Vulnerabilities Using LLM and Static Analysis. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE*.
[57] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1398–1409.
[58] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 819–831.
[59] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. 2024. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817* (2024).
[60] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. 2022. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2022).
[61] Mingxi Ye, Xingwei Lin, Yuhong Nan, Jiajing Wu, and Zibin Zheng. 2024. Midas: Mining Profitable Exploits in On-Chain Smart Contracts via Feedback-Driven Fuzzing and Differential Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 794–805.
[62] Mingxi Ye, Yuhong Nan, Zibin Zheng, Dongpeng Wu, and Huizhong Li. 2023. Detecting State Inconsistency Bugs in DApps via On-Chain Transaction Replay and Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 298–309.
[63] Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. [n. d.]. PATCHAGENT: A Practical Program Repair Agent Mimicking Human Expertise. In *USENIX Security 2025*.
[64] Jingwen Zhang, Yuhong Nan, Wei Li, Kaiwen Ning, Zewei Lin, Zitong Yao, Yuming Feng, Weizhe Zhang, and Zibin Zheng. 2025. Finding Insecure State Dependency in DApps via Multi-Source Tracing and Semantic Enrichment. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1529–1540.
[65] Jingwen Zhang, Zibin Zheng, Yuhong Nan, Mingxi Ye, Kaiwen Ning, Yu Zhang, and Weizhe Zhang. 2025. SmartReco: Detecting Read-Only Reentrancy via Fine-Grained Cross-DApp Analysis. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. 1–12.
[66] Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. 2020. EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*. 116–126.
[67] Wuqi Zhang, Zhuo Zhang, Qingkai Shi, Lu Liu, Lili Wei, Yepang Liu, Xiangyu Zhang, and Shing-Chi Cheung. 2024. Nyx: Detecting Exploitable Front-Running Vulnerabilities in Smart Contracts. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 146–146.
[68] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying exploitable bugs in smart contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 615–627.
[69] Zibin Zheng, Jianzhong Su, Jiachi Chen, David Lo, Zhijie Zhong, and Mingxi Ye. 2024. DAppSCAN: Building Large-Scale Datasets for Smart Contract Weaknesses in DApp Projects. *IEEE Transactions on Software Engineering* (2024).

Received 2025-09-11; accepted 2026-03-24