AdvSCanner: Generating Adversarial Smart Contracts to Exploit Reentrancy Vulnerabilities Using LLM and Static Analysis

Yin Wu
Xi'an Jiaotong University
China
wuyin@stu.xjtu.edu.cn

Xiaofei Xie
Singapore Management University
Singapore
xfxie@smu.edu.sg

Chenyang Peng
Xi'an Jiaotong University
China
pcy3123157003@stu.xjtu.edu.cn

Dijun Liu
China Mobile Chengdu Institute of Research and Development
China
liudijun@chinamobile.com

Hao Wu
Xi'an Jiaotong University
China
emmanuel_wh@stu.xjtu.edu.cn

Ming Fan
Xi'an Jiaotong University
China
mingfan@mail.xjtu.edu.cn

Ting Liu
Xi'an Jiaotong University
China
tingliu@mail.xjtu.edu.cn

Haijun Wang*
Xi'an Jiaotong University
China
haijunwang@xjtu.edu.cn

ABSTRACT

Smart contracts are prone to vulnerabilities, with reentrancy attacks posing significant risks due to their destructive potential. While various methods exist for detecting reentrancy vulnerabilities in smart contracts, such as static analysis, these approaches often suffer from high false positive rates and lack the ability to directly illustrate how vulnerabilities can be exploited in attacks.

In this paper, we tackle the challenging task of generating ASCs for identified reentrancy vulnerabilities. To address this difficulty, we introduce AdvSCanner, a novel method that leverages the Large Language Model (LLM) and static analysis to automatically generate adversarial smart contracts (ASCs) designed to exploit reentrancy vulnerabilities in victim contracts. The basic idea of AdvSCanner is to extract attack flows associated with reentrancy vulnerabilities using static analysis and utilize them to guide LLM in generating ASCs. To mitigate the inherent inaccuracies in LLM outputs, AdvSCanner incorporates a self-reflection component, which collects compilation and attack-triggering feedback from the generated ASCs and refines the ASC generation if necessary. Experimental evaluations demonstrate the effectiveness of AdvSCanner, achieving a significantly higher success rate (76.41%) compared to baseline methods, which only achieve 6.92% and 18.97%, respectively. Furthermore, a case study illustrates that AdvSCanner can greatly reduce auditing time from 24 hours (without assistance) to approximately 3 hours when used during the auditing process.

*Corresponding author

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695482

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Reentrancy Vulnerability, Code Generation, Large Language Model, Smart Contract, Static Analysis

ACM Reference Format:
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 *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695482

1 INTRODUCTION

In recent years, blockchain technology has increasingly expanded its capacity to support the Web3 and fintech sectors, primarily due to its ability to support Turing-complete smart contracts. Ethereum, the most popular open-source platform within the blockchain ecosystem, facilitates the deployment of smart contracts to support decentralized applications, digital currencies, and chain management and insurance.

As the value of Ethereum has continued to rise, contracts with vulnerabilities have increasingly attracted the attention of hackers seeking unfair profits, and numerous hacks have resulted in substantial financial losses [1–3], with reentrancy attacks being particularly notable. A significant instance of a reentrancy attack, known as the DAO attack [4], occurred in Ethereum's history, leading to a hard fork into Ethereum Classic (ETC) and Ethereum (ETH) [5]. Since then, as the Ethereum platform and its smart contracts have evolved, so have the tactics of attackers. Very recently, the decentralized exchange Predy Finance [6] on the Arbitrum chain was attacked by hackers, exploiting a reentrancy vulnerability to drain approximately $464k from the liquidity pool. Besides, once

smart contracts are deployed on the blockchain, they cannot be altered due to the immutability characteristic of blockchain technology. Therefore, it is crucial to detect the vulnerabilities in smart contracts, particularly reentrancy vulnerabilities, before they are deployed.

Reentrancy vulnerabilities in smart contracts allow attackers to exploit callback flaws in function re-invocations, enabling repeated entries into the vulnerable contracts [7]. To counteract these reentrancy attacks, several static analysis tools, including Slither [8], SmartCheck [9], Securify [10], and CSA [11], have been developed. These tools aim to detect vulnerabilities in smart contracts without executing the code. However, static analysis often produces numerous false positives, particularly concerning reentrancy vulnerabilities [12]. Even when a reentrancy vulnerability is correctly identified, developers face challenges in confirming its presence and understanding the underlying attack principles. By exploiting reentrancy vulnerabilities, researchers can not only confirm the presence of these vulnerabilities but also illustrate their potential risks and elucidate the underlying attack mechanisms. In this paper, we focus on generating adversarial contracts that can be used to exploit reentrancy vulnerabilities.

Recent studies have concentrated on directly exploiting vulnerabilities by constructing attack transactions [13, 14]. However, crafting successful attack transactions poses significant challenges, as they require precise sequencing and specific operations. Attacks often fail if the transactions are unordered, terminate prematurely, or incorporate logical errors, with no universally effective method to generate effective attack transactions for various vulnerable smart contracts.

An alternative approach to exploit the reentrancy vulnerability involves creating an adversarial smart contract (ASC) designed to initiate these attack transactions. Currently, the development of such adversarial contracts is predominantly a manual process, which is labor-intensive and time-consuming. This process requires a deep understanding of the targeted vulnerable contract and the strategic design of an adversarial contract to initiate the attack. For example, attackers must carefully manage external calls and the status checks and updates of the contract. This paper aims to explore the generation of ASCs, specifically to exploit the reentrancy vulnerability in targeted smart contracts.

The emergence of Large Language Models (LLMs) has marked a significant advancement in problem-solving capacities, particularly in our understanding and application of prompt engineering[15–18]. These models, including the chat generative pre-trained transformer (ChatGPT) [19], have shown promising applications in code understanding and generation [20–22], potentially enabling the creation of ASCs to exploit reentrancy vulnerabilities. However, recent studies [11, 23] reveal that LLMs often struggle to generate effective ASCs when confronted with vulnerable code. This issue primarily stems from the LLMs' training on general knowledge rather than specialized vulnerability-focused knowledge. As a result, LLMs lack a deep understanding of smart contract behavior in the absence of actual code execution. Additionally, they are prone to generating incorrect or non-compliant code due to "hallucinations" in their code understanding capabilities, which can derail the success of attack strategies.

In this paper, we introduce AdvSCanner, a novel method based on LLM for automatically generating ASCs. AdvSCanner enhances the LLM by integrating it with static analysis to provide more precise information, enabling the LLM to better comprehend vulnerabilities and construct effective adversarial contracts. To generate the exploits, we first perform an empirical analysis to categorize distinct attack flows associated with reentrancy vulnerabilities. For a given smart contract identified with reentrancy, AdvSCanner employs static analysis to pinpoint a suitable attack flow and gather essential vulnerability information, which are used to construct informative prompts. Subsequently, we utilize the Chain of Thought (CoT) prompting strategy [24–26] to guide the LLM in generating the adversarial contract. To address the issues of hallucination and potential inaccuracies in the generated output, we also implement a self-reflection mechanism. This mechanism validates the effectiveness of the attack launched by the adversarial contract and generates feedback if fails, facilitating the regeneration of the adversarial contract.

We conducted comprehensive experiments to demonstrate the effectiveness of AdvSCanner. We compared AdvSCanner with LLM-based baseline methods, namely LLM_C (where code is input directly into the LLM) and LLM_{CS} (where both code and static analysis are used as input for the LLM), using identical datasets and experimental conditions. AdvSCanner outperformed these baselines, achieving a generation success rate of 76.41%, compared to 6.92% (LLM_C) and 18.97% (LLM_{CS}), respectively. We also calculated the variance to further assess the reliability of our results. Additionally, we analyzed factors influencing efficiency, revealing that static analysis required 0.98 seconds, LLM responses averaged 39.47 seconds per iteration, and the validation process took 16.62 seconds. Subsequently, we further evaluated the contributions of components within AdvSCanner. The results show that, compared to the complete AdvSCanner, the success rate drops to 1.03% without the attack flow and to 44.87% without the self-reflection component. We also examined the impact of various configurations on AdvSCanner's performance, including temperature settings, feedback iterations, and LLM models. We found that moderate temperature settings and increased feedback iterations could significantly improve the success rate, with GPT-4o showing optimal performance. Finally, we evaluated the practical benefits of AdvSCanner in smart contract auditing, showing that AdvSCanner can reduce auditing time from 24 hours to about 3 hours.

The primary contributions of this paper are as follows:

- We propose, to the best of our knowledge, the first tool designed to automatically generate adversarial smart contracts (ASCs) by combining LLM with static analysis.
- To augment the capabilities of LLM and address the shortcomings of traditional static analysis, we have developed a comprehensive and systematic attack flow for reentrancy vulnerabilities, which facilitates the generation of effective contracts through few-shot learning and a self-reflection mechanism.
- We have conducted comprehensive experiments to validate the effectiveness of AdvSCanner. The results show that the adversarial smart contracts generated by our tool achieve a high success rate of up to 76.41%, underscoring the utility of combining LLM with static analysis for the exploit generation.

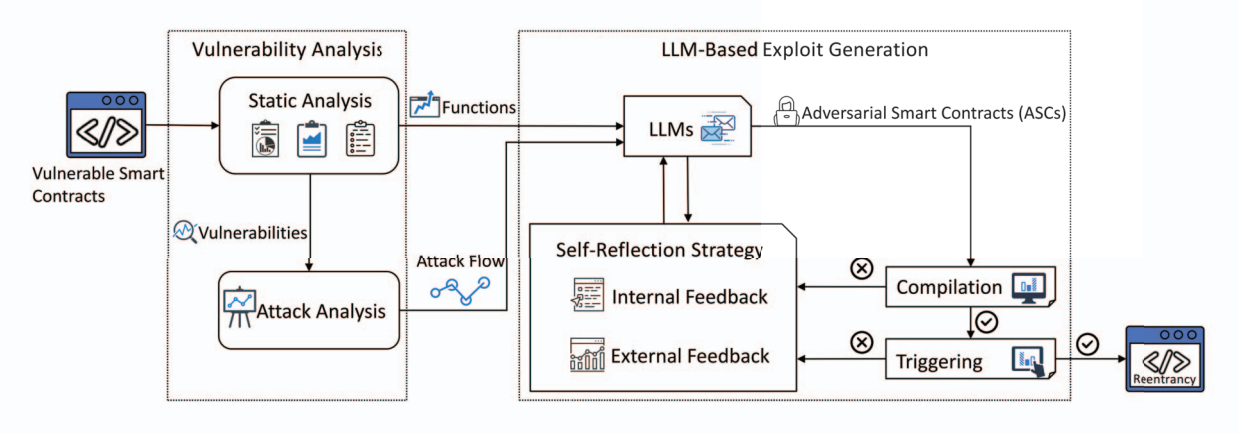

Figure 1: The overall workflow of the AdvSCanner.

2 BACKGROUND

2.1 Reentrancy Vulnerability

Smart contracts [27, 28], as the cornerstone of decentralized applications (dApps) [29] on platforms like Ethereum, are vulnerable to various security threats. The smart contract weakness classification (SWC) system for Ethereum Solidity smart contracts [24] outlines 37 common security vulnerabilities. These vulnerabilities such as integer overflow and underflow (SWC101), outdated compiler version (SWC102), etc., are not within the scope of our analysis as there is no adversarial smart contract for attacking these vulnerabilities. The reentrancy vulnerability (SWC-107) [1, 24, 30, 31] is a critical issue characterized by the mutual invocation between vulnerable smart contracts and adversarial smart contracts through a callback mechanism. In Solidity, the programming language of Ethereum, fallback functions present a challenge as they are triggered when a contract receives Ether, creating opportunities for reentrancy attacks. With the design of new on-chain ERC standard tokens, new reentrancy vulnerabilities are emerging. Poor design in ERC standard tokens and user-defined interfaces further complicates these issues, introducing additional attack vectors. Hackers increasingly exploit this vulnerability to gain unauthorized access and extract unfair profits from affected contracts.

2.2 Static Analysis Tool

Ensuring the security of smart contracts is paramount to the reliability and trustworthiness of decentralized applications. Static analysis tools [32, 33] are crucial for comprehensive security checks, as they analyze code without execution, offering faster analysis and early detection of potential issues compared to dynamic analysis. Several contract analysis tools [34, 35], including Slither [8], SmartCheck [9], and Securify [10], use static analysis techniques to detect vulnerabilities, with Slither being particularly effective in identifying reentrancy issues [17, 36]. However, static analysis tools still struggle with high false-positive rates and detection limitations. Nonetheless, they remain essential for early-stage security checks, providing a foundational layer of defense in the development of secure smart contracts.

2.3 Large Language Models

The development of Large Language Models (LLMs) has revolutionized fields such as natural language processing [37], code generation [20, 21, 38], and automated program repair (APR) [39, 40]. LLMs training typically involves two stages: pre-train and fine-tune. During pre-train, the model learns from large volumes of data, optimizing its parameters and improving its understanding. During fine-tune, the model is trained on task-specific datasets, adjusting its parameters to meet specific requirements. Recently, the "pre-train, fine-tune" approach has evolved into "pre-train, prompt, and predict" [41], where prompt engineering [20, 42, 43] guides the model to perform specific tasks accurately. While LLMs excel at generating human-like code, their inherent randomness can lead to undesired results and weak justification, posing challenges for accurate vulnerability detection and reliable guidance in smart contract security.

3 MOTIVATIONS AND OVERVIEW

In this section, we introduce the challenges and motivations behind integrating LLM with static analysis tools for detecting vulnerabilities, followed by an overview of the design of AdvSCanner.

3.1 Motivation

- **The need for appropriate prompts to generate ASCs.** Direct input of a vulnerable smart contract for generating exploits into LLM typically results in a response like: "*I am sorry, but I can not assist with creating attack contracts or engaging in any activities that could be harmful. If you have any other questions, feel free to let me know.*" This limitation stems from LLM being programmed to adhere to ethical guidelines, which restrict their assistance in creating potentially harmful outputs, such as adversarial smart contracts. Thus, crafting effective prompts that circumvent these ethical constraints remains a challenge.
- **Requirement of attack analysis for generating ASCs.** Successfully exploiting reentrancy vulnerabilities requires a deep understanding of contract behavior. A critical factor is the correct implementation of the fallback function, which allows the malicious contract to take over the control flow and reenter the

victim contract during an attack. If this function is missing or incorrectly implemented, the attack sequence will fail. Furthermore, as ERC standard tokens are designed, additional reentrancy vulnerabilities are emerging. Poor design in both standard ERC tokens and user-defined interfaces can introduce new attack vectors, exacerbating the risk. Hence, concise and targeted attack analysis is essential to equip LLM with the necessary domain knowledge to generate effective ASCs.

- **The necessity of feedback for refining ASCs.** Due to the complexity of the task, it is often difficult for LLM to successfully generate an exploit in a single iteration. Common failures in the generated ASCs can be attributed to issues during compilation or deployment, incomplete or incorrect logic. Addressing these issues requires iterative revisions based on thorough failure analysis, enabling the regeneration of a more effective contract to exploit the identified vulnerabilities.

3.2 Overview

Building on the motivations discussed previously, we have developed AdvSCanner, a cutting-edge framework designed to automatically generate ASCs to exploit reentrancy vulnerabilities in smart contracts. As illustrated in Figure 1, AdvSCanner comprises two main components: vulnerability analysis and LLM-based exploit generation. These components work collaboratively to produce an ASC.

More specifically, for a given smart contract identified with reentrancy vulnerabilities, we initially perform static analysis to gather detailed information about these vulnerabilities. Based on this information, we conduct an attack analysis that aligns the vulnerabilities with specific attack flows, which have been derived from our empirical analysis. Utilizing both the vulnerability details and the identified attack flows, we create CoT prompts that guide the LLM in generating the ASC. We then validate the generated contract from syntactic and semantic perspectives, ensuring it can be successfully compiled (i.e., Compilation Checking) and is capable of exploiting the reentrancy vulnerabilities (Triggering Checking). Once there are any failures, the compilation and triggering checking modules provide relevant feedback. This feedback informs the self-reflection process, which refines the generated smart contract by incorporating the identified failures into the prompts.

4 REENTRANCY ATTACK ANALYSIS

4.1 Vulnerability Analysis

Most static analysis tools can extract reentrancy vulnerability information such as functions and flows from given vulnerable smart contracts. Specifically, we applied the existing static analysis tool to obtain functions leading to modify variables by external calls (e.g., "withdrawAll()" in Figure 2) and functions used to modify variables (e.g., "deposit() / withdrawSome()" in Figure 2) for the further attack analysis. Among them, these flagged functions related to vulnerabilities are from the functions influenced by state changes of external call variables. We refer these functions as *utilizable* functions. Not all flagged functions are necessarily vulnerable to reentrancy attacks as some may have appropriate safeguards or lack conditions conducive to such exploits. Hence, the multiple functions (from static analysis tools) flagged for reentrancy vulnerabilities could cause false positives. This identification of such functions results from state changes of the shared variable after external calls, as such changes are typical indicators of reentrancy vulnerabilities.

Figure 2 shows an example regarding the vulnerability analysis, the static analysis tool, Slither, has discovered the function that triggers vulnerabilities in *"Vulnerable.withdrawAll()"* function (line 1). The reentrancy vulnerabilities occur after external calls that modify state variables within the function. Therefore, the state change of *"balance[msg.sender]"* (line 3) after the external call poses a potential risk for reentrancy vulnerabilities. When an adversarial smart contract is reentered potential reentrancy points before the completion of the previous invocation, it can lead to unexpected behaviors and exploitation by hackers. The static analysis tool identifies these points (lines 7-10), including functions *"Vulnerable.deposit()"*, *"Vulnerable.userBalance(address)"*, *"Vulnerable.withdrawAll()"*, and *"Vulnerable.withdrawSome(uint256)"*. Such functions will be used for the following analysis.

```
1  Reentrancy in Vulnerable.withdrawAll() (00-basic.sol#16-24):
2          External calls:
3          - (success) = address(msg.sender).callvalue: balance[msg.sender]()
           ↪  (00-basic.sol#20)
4          State variables written after the call(s):
5          - balance[msg.sender] = 0 (00-basic.sol#23)
6          - Vulnerable.balance (00-basic.sol#7) can be used in cross function reentrancies:
7          - Vulnerable.deposit() (00-basic.sol#11-13)
8          - Vulnerable.userBalance(address) (00-basic.sol#37-39)
9          - Vulnerable.withdrawAll() (00-basic.sol#16-24)
10         - Vulnerable.withdrawSome(uint256) (00-basic.sol#26-34)
```

Figure 2: Reentrancy vulnerabilities discovered by the static analysis.

4.2 Attack Analysis

To effectively generate an ASC capable of performing a reentrancy attack, it is crucial to understand the potential attack flows stemming from reentrancy vulnerabilities. Such insights directly inform the LLM in generating ASCs. Therefore, we have conducted a detailed analysis to categorize and summarize the different types of attack flows.

4.2.1 General Attack Flow. Reentrancy vulnerabilities often manifest through mutual calls between vulnerable contracts and adversarial contracts, which include callback features akin to recursive calls in traditional programming. As depicted in Figure 3, the attack commences when hackers call the *utilizable* function of the vulnerable contract. Then the operation within the function immediately triggers one of three callback mechanisms in the adversarial smart contract. These callbacks can be categorized into three main types:

(1) Fallback Mechanism (Type A), which is triggered in Solidity smart contracts upon receiving Ether, employs the *"fallback"* or *"receive"* functions.
(2) Hook Function (Type B), which involves the use of hook functions associated with ERC standard tokens to trigger the callback function of the adversarial contract.
(3) User-Defined Interfaces (Type C), which are poorly designed user-defined interfaces that lead to challenges for static analysis due to high rates of false positives and false negatives, which we exclude from our study for its significant uncertainty.

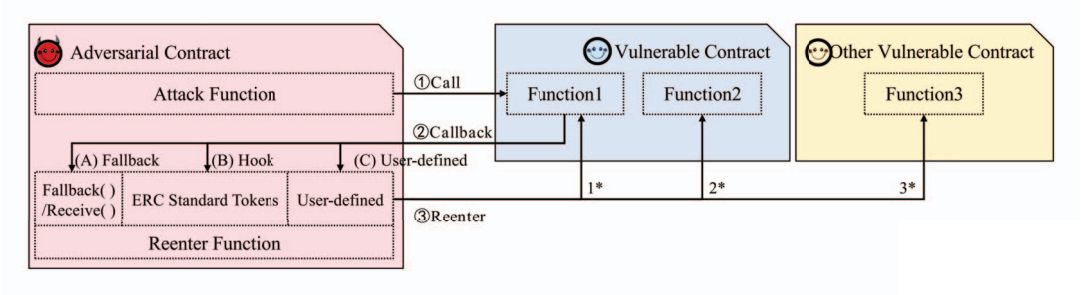

Figure 3: The attack flow of reentrancy vulnerabilities.

Subsequently, hackers strategically execute the callback mechanism to reenter functions in the vulnerable contract or other contract following some reentering paths. Reentrancy attacks typically continuously reenter the vulnerable contracts based on the callback mechanism. Within this setup, we summarize three common reentering paths (1*-3*) for repeated entrance (see Figure 3):

(1) Path 1* (Single Reentrancy) implements the same *utilizable* function of the vulnerable contract.
(2) Path 2* (Cross-Function Reentrancy) involves different *utilizable* functions within the same vulnerable contract.
(3) Path 3* (Cross-Contract Reentrancy) entails executing functions from other vulnerable contracts or contract libraries.

During the process of reentering the vulnerable contracts, these reentrancy attack leverage specific types and paths within and across contracts, enabling attackers to manipulate contract state in unintended ways.

4.2.2 Attack Flow Extraction. Based on the above analysis, we further extract a concrete attack flow from a reentrancy vulnerability, which is further used to construct the prompts. Specifically, from Figure 3, we summarize 6 types of concrete attack flows based on the attack type (i.e., Type A or Type B) and the paths (i.e., Path 1*-3*), which excludes Type C for its significant uncertainty. Figure 4 shows the examples for the 6 concrete attack flows. The first three are the fallback mechanism, involving functions like *"withdrawAll()"* that trigger vulnerabilities when sending Ether to an address, which in turn activates *"fallback()"* function or *"receive()"* function. The last three use ERC standard tokens to inject the callback into the Hook function. Specifically, when a function like *"Mint()"* is called, the receiving contract must implement the *"onERC721Received()"* hook. This hook can execute malicious code, enabling reentrancy by recursively calling the vulnerable function in the victim contract. The loop persists until all variables are updated or a reentrancy limit halts further transactions.

Algorithm 1 shows the algorithm that matches a given vulnerability (V_i) to a type of attack flow (F). The vulnerability has been detected by static analysis tools to gain flagged functions (N_p, N_{c1} and N_{c2}). In the matching methodology, the function call of state variables N_p following *"External call"* are utilized for type-matching, discerning if the callback mechanism is (Type A) Fallback Mechanism or (Type B) Hook Function. If the function call is one of these transfer calls $\Upsilon 1$, *".callvalue()"* (seen in Figure 2 line 3), *".transfer()"*, *".send()"*, or *".delegatecall()"*, the callback mechanism is referred to as Type A. If the function call $\Upsilon 2$ uses ERC standard tokens like

Algorithm 1 Attack Flow Matching

Input: $\{V_i, i = 1, 2, 3...\}$
Output: F
$\{N_p, N_{c1}, N_{c2}\} \leftarrow N(V_i)$
$\Upsilon 1 \leftarrow \{(.callvalue), (.transfer), (.send), (.delegatecall)\}$
$\Upsilon 2 \leftarrow \{ERC_Standard_Tokens\}$
if $N_p \subseteq \Upsilon 1$ **then**
  if $N_{c1} \subseteq N_{c2}$ **then**
    if $N_{c1} == N_{c2}$ **then**
      $F \leftarrow F_1$
    else
      $F \leftarrow F_2 \& F_3$
    end if
  end if
else if $N_p \subseteq \Upsilon 2$ **then**
  if $N_{c1} \subseteq N_{c2}$ **then**
    if $N_{c1} == N_{c2}$ **then**
      $F \leftarrow F_4$
    else
      $F \leftarrow F_5 \& F_6$
    end if
  end if
else
  $F \leftarrow \{N_{c1}, N_{c2}\}$
end if
return F

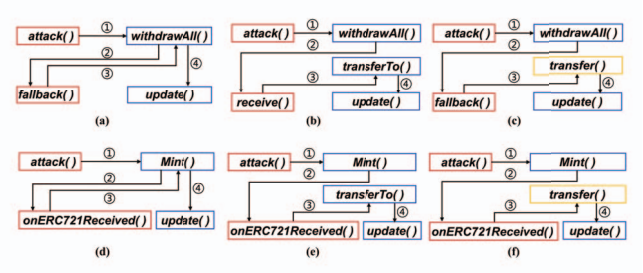

Figure 4: The illustrations after flow information extraction.

"ERC721/ERC777/ERC1155/ERC223/ERC677()", the callback mechanism is identified as Type B. Once Type N_p is determined, functions triggering vulnerabilities N_{c1} (e.g., in Figure 2 line 1) and vulnerabilities N_{c2} (e.g., in Figure 2 lines 7-10) are utilized for function-matching to match the attack flow (i.e., F_1 to F_6). The Paths (1*-3*) in the repeated loop are determined by comparing these functions N_{c1} and N_{c2}. Identical functions indicate Single Reentrancy, while differing functions suggest Cross-Function or Cross-Contract Reentrancy. If the vulnerability does not match any predefined attack flows, static analysis can still provide valuable insights. Flagged functions N_{c1}, N_{c2} from the static analysis can be transmitted to the LLM to aid in providing context, even without specific attack flow information.

4.3 Adversarial Smart Contract Generation

Based on the static analysis on the reentrancy and attack flow, AdvSCanner further leverages the LLM to generate ASCs by providing informative prompts. As shown in Figure 5, next we will introduce our prompt design.

Task Decomposition. Task decomposition enhances efficiency by breaking down complex tasks into manageable steps, thereby improving focus and optimizing task completion. In the context of static analysis, the contract generation problem is divided into three main steps: *defining roles*, *interpreting static analysis results*, and *analyzing attack flow*.

The Chain of Thoughts (CoT). To bypass LLM security protocols that may intercept malicious behavior, we design the adaptive prompts using the CoT approach [24, 44]. This involves combining long-term conversations with few-shot learning using suitable prompts to guide the LLM analyzing the static analysis results and the attack flow, as shown in Figure 5. This method avoids triggering the security filter of LLM, allowing for the generation of ASCs.

Prompt with Attack Flow. Flow information extraction involves identifying the attack flow of reentrancy vulnerabilities, which is critical for prompt engineering, as it directly serves as background information for the LLM. The existing attack flows (F1-F6) obtained from the matching methodology in Algorithm 1 are parsed and abstracted into specific reentrancy prompts (TP1-TP6). The ASC predefined prompt for TP1 is shown in Figure 5, including setting contract addresses in the *"constructor()"*, sending Ether to the vulnerable contract for gas by *"deposit()"*, initiating *"attack()"* transactions to the vulnerable contract, and executing reenters using *"receive()"* or *"fallback()"*. The specific operations are determined based on the flagged functions extracted from the attack flow.

4.4 Self-reflection Strategy for Validation

Given that smart contracts generated by LLM may not always be valid for adversarial purposes, we propose a self-reflection strategy to validate their correctness from two key perspectives: syntax correctness, ensuring successful compilation, and semantic correctness, confirming their ability to trigger a reentrancy attack. If the smart contract fails to meet these criteria, we provide detailed failure feedback to the LLM to inform the regeneration of the contract.

Algorithm 2 shows the algorithm of self-reflection. The appropriate prompt TP_x is input into the LLM *chatbot* $(\cdot)$ to obtain model

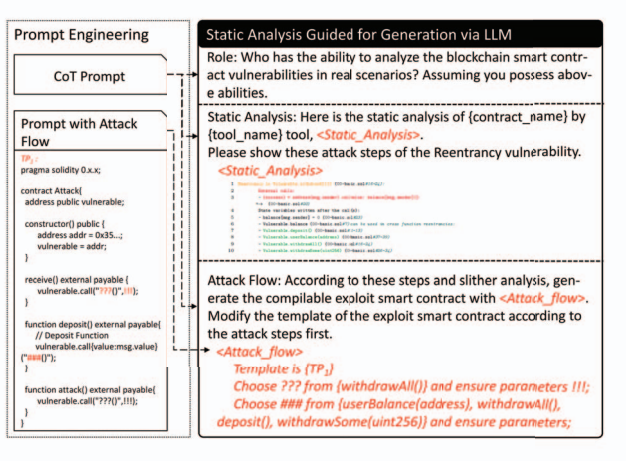

Figure 5: Adaptive prompts for generating adversarial smart contracts.

Algorithm 2 A Feedback Workflow of the AdvSCanner Tool with the Self-reflection Strategy

Input: TP_x, C_s
Output: Λ

$P_{all} \leftarrow TP_x$
$\Phi = 0, \Phi_{error} = \emptyset, a = 1, \text{Max} = 5, \Lambda = 0$
while $\Phi = 0$ & $a \leq \text{Max}$ **do**
  $R_0 \leftarrow chatbot\ (P_{all} \cup \Phi_{error})$ ▷ External Feedback
  $C_m \leftarrow Extract\ (R_0)$
  $\{\Phi, \Phi_{error}\} \leftarrow Compile\ (C_m)$
  $R_1 \leftarrow chatbot\ (\Phi_{error})$ ▷ Self-reflection
  $P_{all} = P_{all} \cup R_1$ ▷ Internal Feedback
  $a = a + 1$
  if $\Phi = 1$ **then**
    $\{\Lambda, \Lambda_{\text{path}}\} \leftarrow Deploy\ (C_m, C_s)$
    if $\Lambda = 0$ **then**
      $R_2 \leftarrow chatbot\ (\Lambda)$ ▷ Self-reflection
      $P_{all} = P_{all} \cup R_2$ ▷ Internal Feedback
      $\Phi = 0, \Phi_{error} = C_s$
    end if
  end if
end while
if $\Lambda = 1$ **then**
  print("Successful Attack")
end if
return Λ

replies R_0. An ASC code C_m is extracted from these replies R_0 via $Extract\ (\cdot)$. Initially, the generated contract C_m is compiled by $Compile\ (\cdot)$. If compilation fails, Φ is set to 0. Error messages Φ_{error} are included in the next round until a compiled contract is generated or the maximum Max attempts are reached. If successful, Φ is set to 1, Φ_{error} is $\emptyset$, and the process proceeds to triggering deployment. Both the generated C_m and source C_s smart contract codes are deployed to analyze the attack result Λ and its attack path Λ_{path}. If triggering is unsuccessful, Φ is set to 0 and Φ_{error} is set to

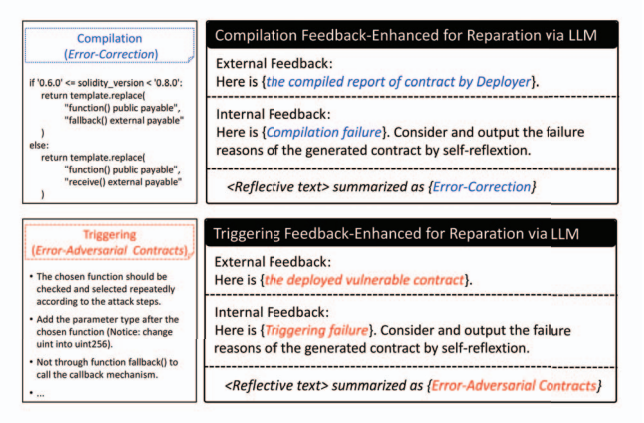

Figure 6: Self-reflection prompts for repairing adversarial smart contracts.

C_s, which re-inputs to the LLM until successful triggering confirms the reentrancy vulnerability $\Lambda_{\text{path}} = 1$ or maximum attempts Max confirm its absence. Compilation and triggering failures provide external feedback, while self-reflection as internal feedback R_1, R_2 enhances subsequent decision-making.

Compilation Feedback Static analysis allows for compilation checks, providing error feedback to the LLM until successful compilation. To minimize feedback iterations and improve efficiency, we employ the self-reflection mechanism Algorithm 2. This mechanism enables the model to self-reflect on issues in the generated ASCs. The reflection results form the reflective text, summarized into error-correction pairs, iteratively refining previous inefficient prompts P_{all}. The reflective text derived from compilation failure feedback includes common compilation errors related to Solidity versions, constructor addresses, and call types. As illustrated in Figure 6, *"fallback"* and *"receive"* functions have specific usage requirements across different Solidity versions. Pre-configuring such common compilation conditions can significantly reduce their occurrence, enhancing the efficiency and accuracy of the contract generation.

Triggering Feedback To simulate real-world attack scenarios, we deploy victim contracts and execute adversarial contracts using the py-evm framework[45] for interactive testing. The primary goal is to verify if the generated ASCs can successfully exploit the victim contracts. Initially, specific hooking opcodes, such as *call*, *staticcall*, and *delegatecall*, are defined to gather their operands for subsequent identification. During the execution of the adversarial contract, we simulately execute each application binary interface (ABI) call and retrieve all operands associated with these hooking opcodes. Next, we extract the operands from the stack to identify the corresponding call address and function signature. This allows us to capture the execution information of the identified function in the victim contract to determine if it can call back into the adversarial contract. By capturing these call trajectories, we analyze the interactions between the victim and adversarial contracts. Finally, we compare the captured call trajectories from the interactive testing with a standard attack flow (i.e., call, callback, reenter) to validate whether the generated adversarial contracts successfully exploit the victim contracts. When interactive testing fails, the failure trajectory is used as external feedback, fed into the LLM for self-reflection to generate internal feedback on the failed attack. Considering the complexity of interactive testing, the failure results consist of the failure trajectory and source vulnerable contracts to mine interaction information between vulnerable and adversarial contracts. The reflective text, illustrated in Figure 6, includes issues such as incorrect parameters, inappropriate function selection, incorrect addresses, and the absence of callbacks. The reflective text is generated after self-reflection to refine the previous inefficient prompts P_{all}, thereby increasing the success rate of the attacks.

5 EXPERIMENTAL EVALUATION

We aim to answer the following research questions.

- **RQ1**: How effective is AdvSCanner compared to the baselines for generating adversarial smart contracts?
- **RQ2**: What are the contributions of different components of AdvSCanner in improving generation effectiveness?
- **RQ3**: How does AdvSCanner perform with different configurations?
- **RQ4**: What is the helpfulness of the AdvSCanner tool to the manual audit in confirming reentrancy vulnerabilities?

5.1 Experimental Setup

5.1.1 Dataset. To evaluate the effectiveness of AdvSCanner in generating adversarial smart contracts, we collected a diverse set of benchmarks based on four criteria:

#1 **Open-source and Peer-reviewed Dataset:** The dataset is sourced from widely-used or peer-reviewed datasets that are open-source and accepted in the research community.
#2 **Marked as Reentrancy Vulnerability:** The dataset includes contracts explicitly marked as containing reentrancy vulnerabilities.
#3 **Detection by Static Analysis Tool:** Reentrancy vulnerabilities in the dataset can be detected by traditional static analysis tools.
#4 **Fully Functional Characteristics:** Only contracts with complete functionality are included, as partial-function contracts cannot support attack verification experiments.

As shown in Table 1, we collected smart contracts with reentrancy vulnerabilities from the following sources: ERAP (Eth Reentrancy Attack Patterns) [7, 46], ESC (Ethereum Smart Contract) [47, 48], Smartbugs (smartbugs-curated) [34, 49], SolidiFI-Benchmark [50, 51], Reentrancy Study Data (RSD) [12, 52], BlockWatchdog [36, 53], ATR (All Things Reentry) [54], and SSE (Solid Security by Example) [55]. The total refers to the number of smart contracts in these datasets. For example, among the 185 smart contracts in the ESC dataset, only 53 are labeled as vulnerable reentrancy. Among these labeled vulnerable contracts, only 16 can be identified by static analysis tools and possess fully functional characteristics. Finally, according to the aforementioned criteria, we filtered out ineligible and duplicate smart contracts, resulting in a total of 78 unique smart contracts (14 are duplicate.) with reentrancy vulnerabilities [1].

[1] https://figshare.com/s/3f864a6e7c6245ad9704

Table 1: The selection criteria of datasets.

Dataset	Total	Criteria				Selection
		#1	#2	#3	#4	
ERAP[7]	6	6	6	6	6	**6**
ESC[47]	185	185	53	16	16+	**16**
Smartbugs[34]	31	31	31	31	28	**28(14)**
SolidiFI[50]	50	50	50	50	0	**0**
RSD[12]	40	40	40	38	34	**34**
BlockWatchdog[36]	15	15	0	15	15	**0**
ATR[54]	7	7	7	5	7	**5**
SSE[55]	4	4	4	3	4	**3**
Total	338					**78**

The experiments were conducted on an Ubuntu 22.04 system equipped with an Intel Core i7-9750H CPU @ 2.60GHz (6 cores and 12 threads), with a 12 MB L3 cache and 16 GB of RAM.

5.1.2 Baselines. To the best of our knowledge, there are currently no techniques specifically targeting the automatic generation of adversarial smart contracts to exploit reentrancy vulnerabilities. To address this gap, we have developed two baselines to evaluate the effectiveness of AdvSCanner. Since AdvSCanner is powered by LLM, these baselines also leverage LLM-based techniques for vulnerability analysis and ASCs generation.

The first baseline, LLM_C, directly inputs the vulnerable contract code, embedding the "chain-of-thought" process for detecting and exploiting reentrancy vulnerabilities into the prompt, thereby enabling the LLM to generate ASCs. The second baseline, LLM_{CS}, builds on LLM_C by additionally incorporating static analysis results as input for the LLM. These two baselines fully utilize the capabilities of LLM and static analysis, providing a comprehensive foundation for evaluating the effectiveness of AdvSCanner.

In our experiments, we employ Slither [8] as our primary static detection tool due to its effectiveness in identifying reentrancy vulnerabilities. Other static detection tools that can reliably detect such vulnerabilities are also suitable for this purpose. For the generation of ASCs, we default to *GPT-3.5-turbo*, leveraging its advanced language modeling capabilities, with all experiments assuming its use unless otherwise specified. In all experiments, the approach involves running five trials and averaging the results to account for variability, except for those specifically focused on configuration analysis, where alternative LLMs are explored to ensure comprehensive evaluation.

5.2 RQ1: Effectiveness

To evaluate the effectiveness of AdvSCanner in generating adversarial smart contracts, we conducted comparative analysis against two baseline techniques. The findings, outlined in Table 2, demonstrate the significant superiority of AdvSCanner, achieving an average success rate of 76.41% across five trials. In contrast, LLM_C and LLM_{CS} exhibited notably lower success rates of 6.92% and 18.97%, respectively. Furthermore, the corresponding statistical variance in results for LLM_C, LLM_{CS}, and AdvSCanner were 2.37×10^{-4}, 4.50×10^{-3}, and 3.02×10^{-4}, respectively. These statistics demonstrate a more stable performance by AdvSCanner and LLM_C across the trials, which depend on the model itself while weakening the impact of other analyses.

Table 2: Comparisons of baselines and the AdvSCanner.

Dataset	LLM_C	LLM_{CS}	AdvSCanner
ERAP	13.33%	16.67%	50.00%
ESC	6.25%	26.35%	81.25%
Smartbugs	2.86%	10.00%	67.14%
RSD	5.88%	20.59%	90.00%
ATR	16.00%	8.00%	40.00%
SSE	13.33%	26.67%	53.33%
Average	6.92%	18.97%	76.41%
Variance	2.37×10^{-4}	4.50×10^{-3}	3.02×10^{-4}

Table 3: Performances of the AdvSCanner.

Dataset	Static Analysis	LLM Response	Validation
ERAP	0.89	15.49	15.69
ESC	1.09	52.75	17.30
Smartbugs	0.95	35.66	16.15
RSD	0.88	46.60	17.09
ATR	1.24	11.38	13.66
SSE	0.88	12.29	16.79
Average	0.98	39.47	16.62

The outcomes of contract generation reveal two types of failures: compilation failures and triggering failures, inspiring the adoption of a self-reflection validation strategy to address both. The overall accuracy is presented in Table 2, with detailed statistics provided in the following. LLM_C encountered a total of 363 failures (93.08%), consisting of 32 compilation failures (8.21%) and 323 triggering failures (84.87%). In LLM_{CS}, we observed 316 failures (81.03%), including 67 compilation failures (17.18%) and 249 triggering failures (63.85%). Conversely, AdvSCanner experiments resulted in 92 failures (23.59%), with 10 compilation failures (2.56%) and 82 triggering failures (21.03%). This superior performance is attributed to AdvSCanner's advanced capabilities in accurately analyzing vulnerabilities and crafting precise adversarial contracts, effectively reducing both compilation and triggering failures.

Following the analysis of success rates, we also examined the factors influencing efficiency in Table 3, which include static analysis, LLM responses, the number of iterations, and the validation process. After the analysis, we found that the average times for static analysis, LLM responses (per iteration), and the validation process are 0.98s, 39.47s, and 16.62s, respectively, which are acceptable.

Answer to RQ1: AdvSCanner outperforms the two baselines with a success rate of 76.41% compared to 6.92% and 18.97%, demonstrating the effectiveness of our approach in generating adversarial smart contracts. AdvSCanner significantly outperforms baseline techniques in reducing compilation and triggering failures for contract generation, with its efficiency further validated through comparative performance assessments in vulnerability analysis and adversarial contract crafting.

Table 4: Evaluations of different components in AdvSCanner.

Dataset	A_{NoAF}	A_{NoSR}	AdvSCanner
ERAP	0%	30.00%	50.00%
ESC	0%	47.50%	81.25%
Smartbugs	0%	27.14%	67.14%
RSD	2.35%	55.29%	90.00%
ATR	0%	48.00%	40.00%
SSE	0%	20.00%	53.33%
Average	1.03%	44.87%	76.41%
Variance	9.20×10^{-5}	1.62×10^{-2}	3.02×10^{-4}

Table 5: Results of different temperatures in AdvSCanner.

Dataset	Temperature				
	0.3	0.7	1	1.3	1.7
ERAP	50.00%	50.00%	50.00%	50.00%	0%
ESC	80.00%	80.00%	81.25%	77.50%	0%
Smartbugs	62.86%	60.00%	67.14%	42.86%	0%
RSD	86.47%	85.88%	90.00%	80.59%	0%
ATR	40.00%	40.00%	40.00%	40.00%	0%
SSE	40.00%	33.33%	53.33%	53.33%	0%
Average	73.33%	72.31%	76.41%	67.18%	0%
Variance	2.24×10^{-4}	4.34×10^{-4}	3.02×10^{-4}	1.42×10^{-3}	0

5.3 RQ2: Ablation Study

The experiments assess the contributions of distinct components within AdvSCanner in Table 4, specifically examining the influence of the attack flow and self-reflection components on the success rate of generating ASCs.

AdvSCanner, when lacking the attack flow component (designated as A_{NoAF}), represents a scenario where pertinent attack flow information is excluded. In A_{NoAF} experiments, the success rate is merely 1.03% and its variance is 9.20×10^{-5}, emphasizing the pivotal role of attack flow in adversarial contract generation. The absence of this information drastically diminishes success rates due to the lack of inherent knowledge regarding attack flows. Without such insight, LLM would attempt to explore all conceivable attack flows to exploit reentrancy vulnerabilities, rendering the process inefficient. Thus, the extraction of comprehensive attack flows via static analysis proves imperative for LLM to generate effective adversarial contracts.

Conversely, AdvSCanner without the self-reflection component (denoted as A_{NoSR}) achieves a success rate of 44.87% and a variance of 1.62×10^{-2}. While the presence of attack flow information offers critical guidance for LLM to discern attack flows, the self-reflection component aids LLM in overcoming compiling errors and discovering novel pathways for adversarial contract generation. A_{NoSR}, lacking self-reflection, demonstrates limited understanding of errors, resulting in repetitive mistakes and insufficient corrective measures. Integrating self-reflection into the analysis process can mitigate error iterations, thereby enhancing the likelihood of success within a restricted number of attempts. This underscores how the incorporation of self-reflection strategies within AdvSCanner facilitates iterative refinement, leading to improved success rates by learning from past errors.

Answer to RQ2: The ablation experiments on adversarial contract generation demonstrate the critical importance of attack flow information and the self-reflection mechanism in AdvSCanner. Compared to the AdvSCanner, removing the attack flow reduces the success rate to 1.03%, while omitting the self-reflection component to 44.87%.

5.4 RQ3: Configurations Analysis

In this section, we investigate the impact of different configurations on AdvSCanner, focusing on three influential factors: temperature settings in the LLM, the number of feedback iterations, and the various LLMs employed.

Initially, we varied the temperature settings from 0 to 2, specifically selecting values of 0.3, 0.7, 1, 1.3, and 1.7, while using the *GPT-3.5-turbo* model with the number of feedback iterations set to 5. As shown in Table 5, a temperature of 1.7 resulted in failures due to internal errors, leading to a 0% success rate. These failures were usually accompanied by the following error message: *"openai.error.APIError: Failed to create completion as the model generated invalid Unicode output. Unfortunately, this can happen in rare situations. Consider reviewing your prompt or reducing the temperature of your request"*. Within the 0.3 to 1.3 range, the success rates were 73.33%, 72.31%, 76.41%, and 67.18%, respectively. These findings suggest that lower to moderate temperature settings generally yield higher success rates. The temperature setting controls result diversity; higher temperatures increase randomness, fostering creativity but potentially leading to deviations or nonsensical outputs. For tasks requiring high accuracy in code generation, lower temperatures, particularly those close to 1 (≤ 1) are preferable, producing less diverse but more precise and reliable outputs, ensuring the generated smart contract code compiles successfully and executes the intended attack effectively.

Subsequently, we investigate the impact of varying the number of feedback iterations by configuring feedback times at 1, 5, and 10, while maintaining the temperature at 1 and employing GPT-3.5-turbo. The outcomes delineated in Table 6 exhibit a discernible trend, indicating a notable enhancement in success rates with an increased number of feedback iterations. Specifically, AdvSCanner yielded a success rate of 55.64% with one feedback time, whereas 5 and 10 feedback times yielded higher success rates of 76.41% and 83.85%, respectively. These findings underscore the pivotal role of iterative feedback in augmenting the effectiveness of adversarial contract generation, notwithstanding the heightened computational overhead. Considering this trade-off, we opt for 5 feedback iterations in the experiments to strike a balance between effectiveness and computational resources.

Lastly, we compare the performance of three types of LLMs: GPT-3.5-turbo, GPT-4, and GPT-4o, with the temperature to 1 and feedback times to 5. As illustrated in Table 7, GPT-4o achieve identical performance, yielding a success rate of 81.54%, while surpassing GPT-4, which achieves a success rate of 80.26%. These findings underscore the influence of LLMs in generating adversarial contracts to exploit reentrancy vulnerabilities. Due to the enhancements in

Table 6: Influences of feedback times on AdvSCanner.

Dataset	Feedback Times		
	1	5	10
ERAP	40.00%	50.00%	**66.67%**
ESC	68.75%	81.25%	**86.25%**
Smartbugs	37.14%	67.14%	**77.14%**
RSD	62.35%	90.00%	**94.12%**
ATR	40.00%	40.00%	**44.00%**
SSE	53.33%	53.33%	**86.67%**
Average	55.64%	76.41%	**83.85%**
Variance	2.93×10^{-3}	3.02×10^{-4}	1.05×10^{-4}

Table 7: Comparisons of generation performance between three ChatGPT versions of LLMs.

Dataset	Models		
	gpt-3.5-turbo	gpt-4	gpt-4o
ERAP	50.00%	63.33%	63.33%
ESC	81.25%	86.25%	85.00%
Smartbugs	67.14%	71.43%	80.00%
RSD	90.00%	90.00%	91.18%
ATR	40.00%	48.00%	48.00%
SSE	53.33%	66.67%	53.33%
Average	76.41%	80.26%	81.54%
Variance	3.02×10^{-4}	1.30×10^{-2}	2.37×10^{-4}

model architecture and contextual understanding in GPT-4o compared to GPT-3.5, our automatic adversarial contract generation yielded better results.

Answer to RQ3: The configurations of the LLM significantly influence the effectiveness of AdvSCanner. The temperature setting of the LLM notably impacts the success rate, with optimal performance achieved at lower to moderate temperatures (≤ 1) and a peak success rate of 76.41% at a temperature of 1. Increasing the number of feedback iterations further enhances success rates. Additionally, more advanced models deliver superior performance, demonstrating the impact of improvements in model architecture and contextual understanding on outcomes.

5.5 RQ4: Case Study

To assess the assistance of AdvSCanner in auditing real-world smart contracts for reentrancy vulnerabilities, we conducted a comparative user study. The experiments were structured around varying levels of assistance provided to auditors: Expert Only (no assistance), Expert with Static Analysis, and Expert with AdvSCanner. The subject of our study was *Thunder Brawl*[56], a real-world vulnerable project on the BNB Chain, consisting of 1887 lines of code. Although AdvSCanner did not successfully generate an adversarial contract that exploited the reentrancy vulnerability in this complicated project, it did produce an adversarial contract framework that significantly aided auditors in developing effective exploits. We recruited three auditors with at least one year of auditing experience

```solidity
pragma solidity ^0.8.1;

interface IERC721 {
    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external returns (bytes4);
}

contract Attacker {
    HouseWallet houseWallet =
    ↪ HouseWallet(0xae191Ca19F0f8E21d754c6CAb99107eD62B6fe53);
    function attack() public {
        houseWallet.claimReward(gameId, add, _amount,
        ↪ _rewardStatus, _x1, name1, _add);
    }
    function onERC721Received(
        address _operator,
        address _from,
        uint256 _tokenId,
        bytes calldata _data
    ) external payable returns (bytes4) {
        houseWallet.claimReward(gameId, add, _amount,
        ↪ _rewardStatus, _x1, name1, _add);
        return this.onERC721Received.selector;
    }
}
```

Figure 7: The adversarial contract provided by AdvSCanner in study case.

and extensive expertise in developing, deploying, and analyzing contracts with reentrancy vulnerabilities.

The first auditor, proficient in Solidity and reentrancy vulnerabilities, performed the development without any assistance, requiring approximately 24 hours to implement the adversarial contract. The second auditor, assisted by static analysis, spent about 6 hours generating a successful adversarial smart contract. The static analysis tool identified four reentrancy vulnerabilities in "House_wallet-vulnerable.sol" and three in "THB_Roulette-vulnerable.sol", including the *onERC721Received()* hook function. Verifying these vulnerabilities and developing an adversarial contract to exploit them was time-consuming, taking the second auditor 6 hours.

Despite the adversarial contract generated by AdvSCanner couldn't successfully exploit the complex reentrancy vulnerability, it provided a fundamental framework and valuable insights into callback mechanisms and attack flows. This significantly aided the third auditor, who built upon the generated contract to develop an effective adversarial contract within just 3 hours. The adversarial contract automatically generated by AdvSCanner, shown in Figure 7, highlights the critical elements required for exploiting vulnerabilities. Specifically, the contract includes the *onERC721Received()* function, which triggers the *claimReward* function in the *HouseWallet* contract, illustrating a potential reentrancy attack vector. If the contract is vulnerable, this results in the *onERC721Received()* callback being invoked, reentering *claimReward* to exploit the reentrancy vulnerability.

The potential challenges in handling real-world complex vulnerabilities include the constraints of static analysis, the LLMs' understanding, and token limits. To improve the ability of AdvSCanner in generating real-world exploits, we could either enhance the static analysis to extract accurate attack flow information from complex vulnerabilities or improve the LLMs' code understanding

capabilities (e.g., by Fine-tuning the model) to understand these complex vulnerabilities and generate suitable attack smart contracts.

Answer to RQ4: For the complicated real-world reentrancy vulnerabilities, AdvSCanner may not always successfully generate an adversarial contract. However, it could still provide a valuable framework that guides the development process, thereby reducing the overall development workload.

5.6 Threats to Validity

Our study faces several threats to validity related to the technologies and methodologies applied. The first likelihood risk is data leakage. Even if data leakage occurred, Table 2 still demonstrates the effectiveness of AdvScanner as the same base model is also used in baselines. Additionally, the inherent randomness of LLMs could yield inconsistent answers for identical inputs, threatening result reliability. Lastly, considering Ethical concerns, we will restrict tool access to authorized users only, such as security researchers and auditors with verified educational email addresses. The complete prompts will not be open-sourced unless the requesters are verified. We may also implement some monitoring and logging of the tool's usage to detect and respond to any suspicious activities. More importantly, we will establish and enforce ethical guidelines and policies for the use of the technology, ensuring it is used solely for defensive and educational purposes.

6 RELATED WORK

6.1 Reentrancy Vulnerability Analysis

Reentrancy vulnerabilities pose a severe threat to smart contract security. Current tools leverage static analysis [8, 57], dynamic analysis [58], and deep learning [58–60] to identify contract weaknesses. Static analysis effectively identifies vulnerabilities but has a high false positive rate, while dynamic analysis can miss issues due to incomplete execution path coverage. Deep learning, though promising, remains in its early stages for vulnerability detection. When it comes to effective ASCs generation for reentrancy vulnerabilities, existing tools face significant limitations. Over time, several tools have been developed, each building upon the limitations of its predecessors, such as such as incomplete transaction flow, inability to handle multiple transactions, and reliance on expert intervention. TEETHER [13] generates vulnerability attacks from binary bytecode but lacks precision due to incomplete modeling of the entire transaction flow. ContraMaster [14] improves on this with Oracle-supported dynamic exploit generation but is limited to modifying a single transaction, failing in case of transaction errors. Revery [61] aids experts in writing exploit programs but cannot generate usable exploit programs independently. These limitations underscore the need for advanced techniques to generate adversarial contracts that effectively exploit reentrancy vulnerabilities, a significant challenge in the field.

6.2 LLM-powered Code Generation

Recent advancements in LLM-powered code generation have been driven by the increasing capabilities of models like GPT. Early research focused on code completion tasks [42], with LLMs suggesting code snippets based on partial input. Studies by Bareiss et al. [21] demonstrated the effectiveness of Transformer models in generating syntactically and semantically correct code. Subsequent research extended this to code generation tasks [20], including automatic code synthesis from natural language descriptions or programming languages. LLMs have also been applied to automated bug detection and repair in software development, leveraging the contextual understanding of LLMs to efficiently identify and fix defects [18, 38, 42]. These studies highlight the potential of LLMs to transform software development, from code generation to smart contract analysis.

7 CONCLUSION

In conclusion, our proposed approach, AdvSCanner, demonstrates promising results in the generation of adversarial smart contracts by combining large language model (LLM) with static analysis techniques. By leveraging static analysis to provide attack flows for reentrancy vulnerabilities and integrating LLM with self-reflection strategies, AdvSCanner achieves a high success rate in generating adversarial smart contracts. Through evaluation, we show that AdvSCanner covers up to approximately 76.41% of successful attacks, highlighting its effectiveness in addressing the challenges of generating adversarial smart contracts for reentrancy vulnerabilities. These findings underscore the potential of our approach in enhancing smart contract security and mitigating the risks associated with reentrancy vulnerabilities. Automatic inference of attack flows is still challenging due to the high dependency on specific vulnerabilities and attacks. AdvSCanner can be updated to extend to other vulnerabilities by systematically summarizing for new attack patterns.

8 ACKNOWLEDGMENTS

This work was partially supported by National Key Research and Development Program of China (2022YFB2703503), National Natural Science Foundation of China (62372367, 62232014, 62272377, 62372368), Shaanxi Province Innovation Fund (QCYRCXM-2022-345), Shaanxi Province Sanqin Talent Introduction Program, the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] Kaitai Zhu, Xingya Wang, Zhenyu Chen, Song Huang, and Junhua Wu. 2023. Evaluating Ethereum Reentrancy Detection Tools via Mutation Testing. In *Proceedings of the 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 545–555.

[2] Wenlong Li, Xizhan Gao, and Ruzhi Xu. 2023. Reentrancy Vulnerability Detection Based on Conv1D-BiGRU and Expert Knowledge. In *Proceedings of the 2023 4th International Conference on Information Science, Parallel and Distributed Systems (ISPDS)*. IEEE, 125–129.

[3] Zexu Wang, Jiachi Chen, Zibin Zheng, Peilin Zheng, Yu Zhang, and Weizhe Zhang. 2024. Unity is Strength: Enhancing Precision in Reentrancy Vulnerability Detection of Smart Contract Analysis Tools. *arXiv preprint arXiv:2402.09094* (2024).

[4] 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.

[5] Dmytro Kaidalov, Lyudmila Kovalchuk, Andrii Nastenko, Mariia Rodinko, Oleksiy Shevtsov, and Roman Oliynykov. 2017. Ethereum classic treasury system proposal. *IOHK RESEARCH REPORT* (2017).

[6] Accessed: May, 2024. PredyFinance. https://x.com/Phalcon_xyz/status/1790307019590680851.

[7] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934* (2018).

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

[9] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*. 9–16.

[10] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 67–82.

[11] Kaixuan Li, Yue Xue, Sen Chen, Han Liu, Kairan Sun, Ming Hu, Haijun Wang, Yang Liu, and Yixiang Chen. 2024. Static Application Security Testing (SAST) Tools for Smart Contracts: How Far Are We? *arXiv preprint arXiv:2404.18186* (2024).

[12] Zibin Zheng, Neng Zhang, Jianzhong Su, Zhijie Zhong, Mingxi Ye, and Jiachi Chen. 2023. Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 295–306.

[13] Johannes Krupp and Christian Rossow. 2018. {teEther}: Gnawing at ethereum to automatically exploit smart contracts. In *Proceedings of the 27th USENIX security symposium (USENIX Security 18)*. 1317–1333.

[14] Haijun Wang, Ye Liu, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. 2020. Oracle-supported dynamic exploit generation for smart contracts. *IEEE Transactions on Dependable and Secure Computing* 19, 3 (2020), 1795–1809.

[15] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv:2308.10620 [cs.SE]

[16] Xin Liu, Yuan Tan, Zhenghang Xiao, Jianwei Zhuge, and Rui Zhou. 2023. Not the end of story: An evaluation of chatgpt-driven vulnerability description mappings. In *Proceedings of the Findings of the Association for Computational Linguistics: ACL 2023*. 3724–3731.

[17] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, et al. 2024. UniLog: Automatic Logging via LLM and In-Context Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[18] Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael Lyu. 2024. Domain knowledge matters: Improving prompts with fix templates for repairing python type errors. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[19] Shihao Xia, Shuai Shao, Mengting He, Tingting Yu, Linhai Song, and Yiying Zhang. 2024. AuditGPT: Auditing Smart Contracts with ChatGPT. *arXiv preprint arXiv:2404.04306* (2024).

[20] Peng Li, Tianxiang Sun, Qiong Tang, Hang Yan, Yuanbin Wu, Xuanjing Huang, and Xipeng Qiu. 2023. Codeie: Large code generation models are better few-shot information extractors. *arXiv preprint arXiv:2305.05711* (2023).

[21] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code. *arXiv preprint arXiv:2206.01335* (2022).

[22] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533* (2023).

[23] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2023. When chatgpt meets smart contract vulnerability detection: How far are we? *arXiv preprint arXiv:2309.05520* (2023).

[24] Serge Demeyer, Henrique Rocha, and Darin Verheijke. 2022. Refactoring solidity smart contracts to protect against reentrancy exploits. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*. Springer, 324–344.

[25] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Proceedings of the Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–7.

[26] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[27] Yaqiong He, Hanjie Dong, Huaiguang Wu, and Qianheng Duan. 2023. Formal Analysis of Reentrancy Vulnerabilities in Smart Contract Based on CPN. *Electronics* 12, 10 (2023), 2152.

[28] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469.

[29] Liya Su, Xinyue Shen, Xiangyu Du, Xiaojing Liao, XiaoFeng Wang, Luyi Xing, and Baoxu Liu. 2021. Evil under the sun: Understanding and discovering attacks on ethereum decentralized applications. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*. 1307–1324.

[30] Zhuo Zhang, Yan Lei, Meng Yan, Yue Yu, Jiachi Chen, Shangwen Wang, and Xiaoguang Mao. 2022. Reentrancy vulnerability detection and localization: A deep learning based two-phase approach. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[31] Lejun Zhang, Yuan Li, Ran Guo, Guopeng Wang, Jing Qiu, Shen Su, Yuan Liu, Guangxia Xu, Huiling Chen, and Zhihong Tian. 2023. A novel smart contract reentrancy vulnerability detection model based on BiGAS. *Journal of Signal Processing Systems* (2023), 1–23.

[32] Pengcheng Liu, Yifei Lu, Wenhua Yang, and Minxue Pan. 2023. VALAR: Streamlining Alarm Ranking in Static Analysis with Value-Flow Assisted Active Learning. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1940–1951.

[33] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 544–555.

[34] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 530–541.

[35] Monika Di Angelo and Gernot Salzer. 2019. A survey of tools for analyzing ethereum smart contracts. In *Proceedings of the 2019 IEEE international conference on decentralized applications and infrastructures (DAPPCON)*. IEEE, 69–78.

[36] Shuo Yang, Jiachi Chen, Mingyuan Huang, Zibin Zheng, and Yuan Huang. 2024. Uncover the Premeditated Attacks: Detecting Exploitable Reentrancy Vulnerabilities by Identifying Attacker Contracts. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[37] Zizhuo Zhang and Bang Wang. 2023. Prompt learning for news recommendation. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 227–237.

[38] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

[39] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1162–1174.

[40] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the potential of chatgpt in automated code refinement: An empirical study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[41] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[42] Jialun Cao, Meiziniu Li, Ming Wen, and Shing-chi Cheung. 2023. A study on prompt design, advantages and limitations of chatgpt for deep learning program repair. *arXiv preprint arXiv:2304.08191* (2023).

[43] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839* (2023).

[44] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems* 36 (2024).

[45] Accessed: May, 2024. py-evm. https://github.com/ethereum/py-evm.

[46] Accessed: May, 2024. Eth-Reentrancy-Attack-Patterns. https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns/.

[47] Xingxin Yu, Haoyue Zhao, Botao Hou, Zonghao Ying, and Bin Wu. 2021. Deescvhunter: A deep learning-based framework for smart contract vulnerability detection. In *Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[48] Accessed: May, 2024. Ethereum Smart Contract. https://github.com/MRdoulestar/DeeSCVHunter/tree/main/preprocessing/data/.
[49] Accessed: May, 2024. Smartbugs-Curated. https://github.com/smartbugs/smartbugs-curated/.
[50] 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.
[51] Accessed: May, 2024. SolidiFI-benchmark. https://github.com/DependableSystemsLab/SolidiFI-benchmark/.
[52] Accessed: May, 2024. ReentrancyStudy-Data. https://github.com/InPlusLab/ReentrancyStudy-Data/.
[53] Accessed: May, 2024. BlockWatchdog. https://github.com/shuo-young/BlockWatchdog/.
[54] Accessed: May, 2024. All-Things-Reentrancy. https://github.com/jcsec-security/all-things-reentrancy/.
[55] Accessed: May, 2024. Solidity-Security-by-Example. https://github.com/serial-coder/solidity-security-by-example/.
[56] Accessed: May, 2024. ThunderBrawl. https://x.com/peckshield/status/1575890733373849601.
[57] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.
[58] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 259–269.
[59] Zijun Feng, Yuming Feng, Hui He, Weizhe Zhang, and Yu Zhang. 2023. A bytecode-based integrated detection and repair method for reentrancy vulnerabilities in smart contracts. *IET Blockchain* (2023).
[60] Christoph Sendner, Huili Chen, Hossein Fereidooni, Lukas Petzi, Jan König, Jasper Stang, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2023. Smarter Contracts: Detecting Vulnerabilities in Smart Contracts with Deep Transfer Learning.. In *Proceedings of the NDSS*.
[61] Yan Wang, Wei Wu, Chao Zhang, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2019. From proof-of-concept to exploitable: (One step towards automatic exploitability assessment). *Cybersecurity* 2 (2019), 1–25.