TRACE: Securing Smart Contract Repository Against Access Control Vulnerability

Chong Chen, Lingfeng Bao, *Member, IEEE*, David Lo, *Fellow, IEEE*, Yanlin Wang, *Member, IEEE*, Zhenyu Shan, Ting Chen, *Member, IEEE*, Guangqiang Yin, Jianxing Yu, Zibin Zheng, *Fellow, IEEE*, and Jiachi Chen

***Abstract*—Smart contract vulnerabilities have led to billions of dollars in economic losses. Among these, improper *Access Control*, which allows unauthorized users to execute restricted functions, is particularly prevalent and has caused significant financial damage. Smart contract repositories contain source code, documentation, configuration files, and other artifacts necessary for building and deploying smart contracts. GitHub hosts numerous open-source repositories of this kind, which serve as intermediate artifacts in development and require compilation and packaging to produce deployable contracts. Third-party developers often reference, reuse, or fork code from these repositories during custom development. However, if the referenced code contains vulnerabilities, it can introduce significant security risks. Existing tools for detecting smart contract vulnerabilities are limited in their ability to handle such complex repositories, as they typically require the target contract to be compilable to generate an abstract representation of the program for further analysis. This paper presents TRACE, a tool designed to secure non-compilable smart contract repositories against access control vulnerabilities. TRACE employs LLMs to locate sensitive functions involving critical operations (e.g., *transfer*) within the contract and subsequently completes function snippets into a fully compilable contract. TRACE constructs a function call graph from the abstract syntax tree (AST) of the completed contract. It uses the control flow graph (CFG) of each function as node information. The nodes of the sensitive functions are then analyzed to detect *Access Control* vulnerabilities. Experimental results demonstrate that TRACE outperforms state-of-the-art tools on an open-sourced CVE dataset, detecting 14 out of 15 CVEs. In addition, it achieves 89.2% precision on 5,000 recent on-chain contracts, far exceeding the best existing tool at 76.9%. On 83 real-world repositories, TRACE achieves 87.0% precision, significantly surpassing *DeepSeek-R1*'s 14.3%.**

***Index Terms*—Smart contract, vulnerability detection, access control, LLM, static analysis.**

Received 12 August 2025; revised 20 December 2025; accepted 27 January 2026. Date of publication 6 February 2026; date of current version 20 April 2026. This work was supported in part by the Intelligent Systems Security Laboratory under Grant Q25008*2; in part by the National Natural Science Foundation of China under Grant 62302534, Grant 62332004, and Grant 62276279; in part by Guangdong Basic and Applied Basic Research Foundation under Grant 2025A1515011632 and Grant 2024B1515020032; in part by Zhejiang Province "JianBingLingYan+X" Research and Development Plan (2025C02020); in part by Zhejiang Provincial Science Foundation of China (LQK26F020002); and the Sichuan Provincial Natural Science Foundation for Distinguished Young Scholars (2023NSFSC1963). Recommended for acceptance by J. Wang. *(Corresponding authors: Zhenyu Shan; Ting Chen; Jiachi Chen.)*

Chong Chen is with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310027, China, and also with the Intelligent Transportation and Information Security Laboratory, Hangzhou Normal University, Hangzhou 311121, China (e-mail: chench578@mail2.sysu.edu.cn).

Lingfeng Bao and Jiachi Chen are with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310027, China, and also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou 310013, China (e-mail: lingfengbao@zju.edu.cn; chenjch86@mail.sysu.edu.cn).

David Lo is with the School of Computing and Information Systems, Singapore Management University, Singapore 188065 (e-mail: davidlo@smu.edu.sg).

Yanlin Wang, Jianxing Yu, and Zibin Zheng are with the School of Software Engineering, Sun Yat-sen University, Zhuhai 519082, China (e-mail: wangylin36@mail.sysu.edu.cn; yujx26@mail.sysu.edu.cn; zhzibin@mail.sysu.edu.cn).

Zhenyu Shan is with the Intelligent Transportation and Information Security Laboratory, Hangzhou Normal University, Hangzhou 311121, China (e-mail: 20100119@hznu.edu.cn).

Ting Chen is with the School of Computer Science and Engineering (School of Cyber Security), University of Electronic Science and Technology of China, Chengdu 611731, China, and also with Shenzhen Institute for Advanced Study, UESTC, Shenzhen 518000, China (e-mail: brokendragon@uestc.edu.cn).

Guangqiang Yin is with the School of Computer Science and Engineering (School of Cyber Security), University of Electronic Science and Technology of China, Chengdu 611731, China, and also with Kashi Institute of Electronics and Information Industry, Kashi 844000, China (e-mail: yingq@uestc.edu.cn).

Digital Object Identifier 10.1109/TSE.2026.3660900

I. INTRODUCTION

SMART contracts are self-executing programs that automatically enforce agreements without intermediaries [1]. These contracts operate on decentralized platforms, offering a mechanism for automating transactions and interactions. As a result, they form the backbone of decentralized applications (DApps) [2], enabling functionality such as financial transactions [3], supply chain tracking [4], and digital identity management [5]. By leveraging smart contracts, DApps provide users with trustless, transparent, and immutable services, eliminating reliance on centralized entities.

Despite their advantages, smart contracts face significant security challenges, particularly *Access Control* vulnerability [6]. These issues arise when contracts fail to properly restrict or validate permissions for executing sensitive functions. For example, the *Parity Multi-sig bugs* resulted in a loss of 153,037 Ether (valued at 30 million USD at the time) [7] due to the improper validation of the $msg.sender$'s permissions, which allowed unauthorized users to assume ownership and execute critical operations.

Existing tools for detecting *Access Control* vulnerabilities in smart contracts primarily rely on static and dynamic analysis.

Static analysis examines the source code to identify potential issues [8], [9], [10], [11], while dynamic analysis observes contract behavior in controlled execution environments [60]. However, these tools are primarily designed for analyzing compilable contracts, assuming the availability of a functioning build system. In practice, many open-source smart contract repositories, especially those still under development, archived, or community-maintained, often lack the necessary configuration files, dependency versions, or coherent structures required for successful compilation. This limitation severely restricts the scope of vulnerability detection, leaving large parts of the contract ecosystem unanalyzed.

Third-party developers frequently reuse or fork open-source smart contract repositories from GitHub as a foundation for their own DApps. Yet, these repositories often pose challenges for **automated security analysis**, as most existing tools rely on **compiled artifacts** such as control flow graphs (CFGs) or abstract syntax trees (ASTs) to perform accurate detection. Many of the repositories lack standardized build instructions and contain complex, interdependent contracts written in varying versions of *Solidity* [12]. This makes it difficult to compile, resolve dependencies, and analyze the code without extensive manual setup. Although these repositories serve as developmental artifacts—including source code, documentation, and configurations—their raw form is often incomplete or inconsistent, especially in early-stage or inactive projects. Assessing their security properties can be non-trivial, hindering safe reuse by other developers and delaying secure integration into larger systems.

As outlined in Section II-A, the compilation step is often complex and resource-intensive, which may hinder the scalability and availability of security auditing. Large language models (LLMs) have shown promise in tasks like code generation [13], summarization [14], and completion [15]. However, a previous study found that directly applying LLMs to vulnerability detection often results in low precision [16]. Additionally, the lengthy code in smart contract repositories can overwhelm LLMs, causing distractions from irrelevant context and reducing performance. Some research methods have attempted to combine LLMs with program analysis for smart contract vulnerability detection [17], [18]. *GPTScan* [18], a tool for detecting logical vulnerabilities, combines LLMs with program analysis. It uses LLMs to identify potential vulnerability scenarios and then verifies them through program analysis. However, it still requires compiling the entire smart contract project, making it time-consuming. Additionally, *GPTScan* mainly targets logical vulnerabilities rather than *Access Control* issues. *AChecker* [19] detects access control vulnerabilities through static data-flow analysis but requires fully compilable contracts, limiting its applicability to non-compilable repositories.

To address these limitations, we propose TRACE, a tool designed to secure non-compilable smar**t** contract **r**epositories against **a**ccess **c**ontrol vuln**e**rabilities. The core innovation of TRACE lies in its unique methodology that integrates LLMs with static analysis. Rather than using LLMs for direct vulnerability detection, we employ them as compilation enablers that transform non-compilable code into analyzable complete contracts, followed by static analysis for vulnerability detection. Specifically, TRACE first filters out smart contracts within projects that involve sensitive operations (i.e., selfdestruct, transfer, state variables modification, low-level external contract call). To mitigate the impact of irrelevant code, our approach begins by extracting function snippets that contain these sensitive operations. This extraction is performed using an LLM, leveraging its capability for semantic code analysis. Next, TRACE leverages the LLM to complete the sensitive function snippets into complete smart contracts, enabling their compilation to generate intermediate representations for more precise analysis. If the completed contract remains uncompilable, TRACE submits the error details and source code to the LLM for self-reflection, prompting it to revise the code based on the errors. TRACE constructs a function call graph (FCG) for the completed contracts, where each function is represented as a node and edges denote function calls. Each node is annotated with sensitive tags, indicating whether the function performs sensitive operations. Additionally, TRACE uses *Slither* [8] to generate a function-level control flow graph (CFG) for each node in the FCG. We then defined four types of risky actions that require strict access control and conducted a search for these actions within the FCG. If any such pattern was identified and found to lack proper access control mechanisms, TRACE flagged the presence of a vulnerability.

We evaluated the performance of TRACE using three datasets. The first dataset is derived from related studies [19], [20], which consists of 15 smart contracts with known *Access Control* vulnerability, each associated with a CVE. TRACE successfully identified 14 out of 15 CVEs, outperforming SOTA tool *AChecker* [19], which detected 12 out of 15 CVEs. The second dataset includes 5,000 recent smart contracts from the Ethereum [21]. For this evaluation, we compared TRACE against six baseline tools capable of detecting *Access Control* vulnerability: Slither [8], GPTScan [18], AChecker [19], Securify [11], Mythril [22], Maian [23], and Manticore [60]. TRACE achieved a precision of 89.2%, surpassing all the baseline tools, where the highest precision is 55.9%, with an average of 25.1%. The third dataset comprises 83 real-world smart contract repositories sampled from the *DAppSCAN* dataset [24], containing a total of 3,092 smart contracts. TRACE detected 40 *Access Control* vulnerabilities, achieving a precision of 87.0%. This performance significantly outperforms the best-performing baseline model, *DeepSeek-R1*, which achieved a precision of 14.3%.

Overall, we have made the following contributions.

- To the best of our knowledge, this study is the first to detect *Access Control* vulnerabilities in non-compilable smart contract repositories, while the proposed methods remain applicable to independent smart contracts.
- By integrating LLMs with traditional program analysis techniques, we developed TRACE and evaluated its performance across three datasets from different sources.
- All source code and data related to this study are publicly available at https://github.com/BugmakerCC/Trace to support future research.

II. Background

A. Smart Contracts

Smart contracts are autonomous programs that are deployed and executed on blockchain [1]. During development and maintenance, ***smart contract repositories*** are stored and collaboratively developed on platforms like GitHub, which include ***single smart contract files***, configuration files, and libraries. Each single smart contract file in the repository implements specific functions that are essential to the overall functionality of the entire repository. Once packaged, compiled, and deployed, these projects become ***smart contracts deployed on the blockchain***, where their bytecode is stored under a specific address. ***Decentralized Application (DApp)*** is a software application that runs on blockchain networks, offering users decentralized and trustless services without relying on centralized authorities [2]. Smart contracts deployed on the blockchain serve as the backend of DApps. This lifecycle reflects the evolving roles of smart contracts from development to deployment.

B. Motivation: Why Analyze Non-Compilable Repositories?

Real-world smart contract repositories are often fragmented and lack a unified build system. Developers exploring third-party code or researchers auditing historical or incomplete contracts frequently encounter repositories that cannot be compiled due to version conflicts, unresolved dependencies, or deprecated tools. While building frameworks like *Hardhat* [25] or *Truffle* [26] can ease compilation in controlled environments, they are not always present or maintained in community-shared or forked repositories. In our empirical analysis (see V-B), only 6.0% of repositories from the *DAppSCAN* [24] dataset compiled successfully without manual intervention. This highlights the practical importance of analyzing non-compilable repositories, especially during early-stage audits, third-party integrations, or when revisiting vulnerable legacy code.

Fig. 1 illustrates the process of compiling, auditing, and forking a smart contract repository as a developer. For the repository in Fig. 1, contracts located in the *contracts* folder contain the core logic of the entire project (e.g., *AloeBlend.sol*, *VolatilityOracle.sol*), but they often **depend on contract files from other folders** (e.g., *AloeBlend.sol* depends on *TickMath.sol* in *libraries* folder). In addition to the interdependencies between files, each file may also be associated with a **different *Solidity* version**. For instance, *AloeBlend.sol* requires *Solidity* version 0.8.10 or higher, while *TickMath.sol* requires at least version 0.5.0. Therefore, using a single compiler version may lead to syntax errors, necessitating the use of multiple compiler versions and the ability to switch between them flexibly. Additionally, **external open-source libraries**, such as *OpenZeppelin* [27], are utilized in some parts of the code. Developers must install and configure these libraries in the correct path. Some **development frameworks** like *Hardhat* [25] or legacy tools like *Truffle* [26] are also required to package the entire repository and successfully compile it. Only after completing this setup can the compiled project be analyzed using security tools,

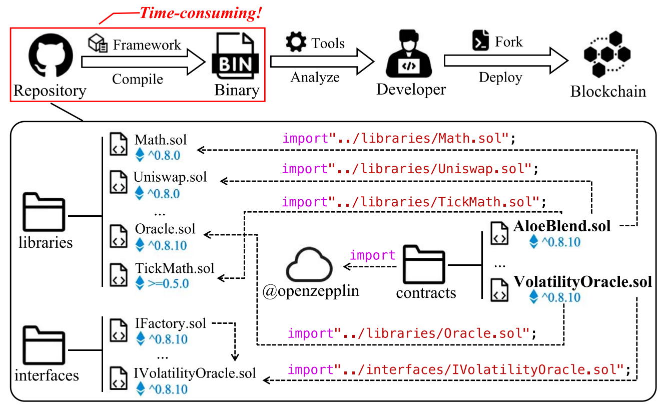

Fig. 1. The process of compiling, auditing, and forking a smart contract repository.

as these tools depend on the intermediate representation of the compiled code to perform security checks. Once a security report is generated, developers can confidently reference and reuse the repository code for custom development tasks, such as forking, before ultimately deploying their finalized contracts on the blockchain.

C. Access Control

Access Control vulnerabilities in smart contracts arise when permission mechanisms fail to restrict unauthorized access to critical operations or resources. Such vulnerabilities can lead to unauthorized fund transfers, manipulation of contract states, or exploitation of administrative privileges. In smart contract repositories, these vulnerabilities can compromise the entire system, causing financial losses and undermining user trust. Ensuring robust access control is crucial for maintaining the reliability of smart contract repositories.

Fig. 2 shows an example of *Access Control* vulnerability in a smart contract repository. The *donate* function in $EtherCharity$ allows any caller to invoke the $selfdestruct$ operation and transfer the contract's balance to an arbitrary $beneficiary$ address. This vulnerability arises because the function lacks any form of authorization checks to verify whether the caller is permitted to perform such a critical operation. As a result, anyone can exploit this function by passing their address as the $beneficiary$, draining all funds from the contract. While the vulnerability logic illustrated in Fig. 2 may appear simple, existing tools struggle to detect it effectively. First, a smart contract repository typically consists of multiple contract files, making it challenging for tools to automatically and accurately identify the main contract. Second, contracts often depend on other contract files within the repository, which may require different versions of the *Solidity* compiler. This necessitates developers preparing multiple compiler versions and seamlessly switching between them. Additionally, if the external libraries relied upon by the contract are not properly loaded, compilation errors can arise. Lastly, redundant code within the contract can distract the attention of LLMs or other machine learning methods, resulting in reduced vulnerability detection performance.

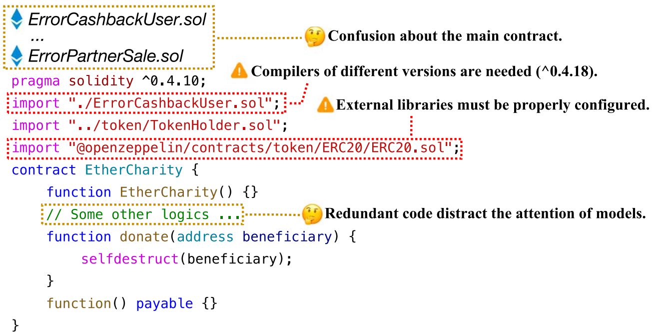

Fig. 2. An example of *access control* vulnerability.

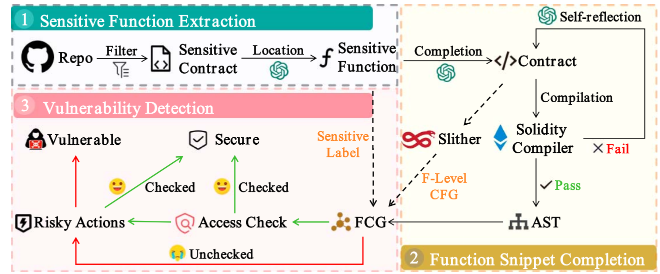

Fig. 3. The overview of TRACE.

D. Static Analysis

Static analysis is a foundational technique in software security that identifies vulnerabilities by analyzing a program's code or bytecode without executing it [28]. In smart contracts, static analysis often requires the code to be fully compilable, as it relies on generating bytecode to disassemble into opcodes and simulate execution in the EVM [29]. Although some tools generate intermediate representations, such as AST, that do not require contracts to be fully compilable, these representations are often insufficiently precise. They typically remain at the syntactic level and fail to capture the logical information involved during program execution. *Slither* [8] is a *Solidity* [12] & *Vyper* [30] static analysis framework written in Python3. It runs a suite of vulnerability detectors, prints visual information about contract details (e.g., contract summary, evm instructions, state variables). In addition, it provides an API to write custom analyses easily, which means users can customize new vulnerability patterns and integrate them into *Slither* for detection.

III. The Design of Trace

Fig. 3 illustrates the architecture of TRACE, comprising three interconnected components. The process begins with **sensitive function extraction**, which identifies functions containing sensitive operations within smart contract repositories, as these are frequent sources of vulnerabilities. The extracted function snippets are then processed through **function snippet completion**, which transforms them into complete smart contracts while maximizing the likelihood of successful compilation. Finally, the completed contracts undergo **vulnerability detection**, where static analysis is performed. This involves constructing a function call graph, enriching each function node with intermediate code representations, and evaluating risky actions and their associated access conditions to detect potential *Access Control* vulnerabilities.

A. Sensitive Function Extraction

Access Control vulnerabilities frequently arise in functions that perform sensitive operations. In this study, we identify four types of such operations, i.e., **Selfdestruct**, **Transfer**, **State Variable Modification**, and **External Contract Call**. The four sensitive operation types are derived from empirical analysis of real-world access control vulnerabilities documented in CVE records and professional audit reports, representing the most prevalent patterns observed in actual exploits. Improper access control of any of these operations can result in serious vulnerabilities. Table I provides detailed definitions of these operations along with examples of potential vulnerability scenarios. We define the functions that contain these sensitive operations as sensitive functions.

To focus our analysis on relevant contracts, we prune the search space by excluding files based on their directory paths. Following standard project structures, we omit files from folders like */interface*, */library*, */util*, */mock*, and */test*. This step is motivated by the fact that these directories typically house code that is non-critical to production security, such as abstract interfaces, trusted external libraries, and testing apparatus, and are therefore excluded from our analysis.

While static analysis is capable of identifying predefined sensitive operations, its efficacy is limited by two key factors. First, pattern matching and lightweight parsing typically require compilable contracts to generate accurate intermediate representations. In non-compilable repositories, these techniques cannot reliably operate. Second, static analysis relies on predefined syntactic patterns that are difficult to define comprehensively. Sensitive operations can be implemented in diverse ways, and exhaustively enumerating all variants is particularly challenging. In contrast, LLMs overcome both limitations. They operate directly on source code, eliminating the need for compilation, and leverage a deeper semantic understanding of the code's logic rather than pre-defined pattern matching. This allows LLMs to locate sensitive functions with greater accuracy and broader applicability, even in non-compilable contracts. Therefore, we utilize LLMs, specifically the *gpt-4o* model's API, to perform sensitive function localization. Building upon prior work [17], we crafted a prompt design as shown in Fig. 4. In the prompt, we specify that the task is to identify sensitive functions within a smart contract and provide a clear definition of what constitutes a sensitive function. Finally, we instruct the model to return the signature of the identified sensitive function. "[CODE]" in the Fig. 4 represents the smart contract source code. After obtaining the function signature, we extract the corresponding function from the contract's source code.

B. Function Snippet Completion

To enable program analysis, we use LLMs to expand extracted sensitive function snippets into complete, compilable

TABLE I
DEFINITION OF SENSITIVE OPERATIONS AND POTENTIAL VULNERABILITY SCENARIOS

Sensitive Operations	**Definitions**	**Scenarios**
Selfdestruct	Remove a smart contract from the blockchain permanently, transferring its balance to a specified address.	The attacker calls the `Selfdestruct` function to transfer all balances from the contract to malicious addresses.
Transfer	Send cryptocurrency from a smart contract to a specified address.	The attacker transfers the balance from the contract or Externally Owned Account (EOA) to malicious addresses.
External Contract Call	Invoke a function in another contract outside the current contract.	The attacker creates an external contract with malicious logic and invokes it in the function.
State Variable Modification	Alter a contract's persistent storage variables, which can impact the contract's behavior and state.	The attacker modifies critical state variables in the contract, such as `balance`, disrupting its ability to operate according to the intended logic.

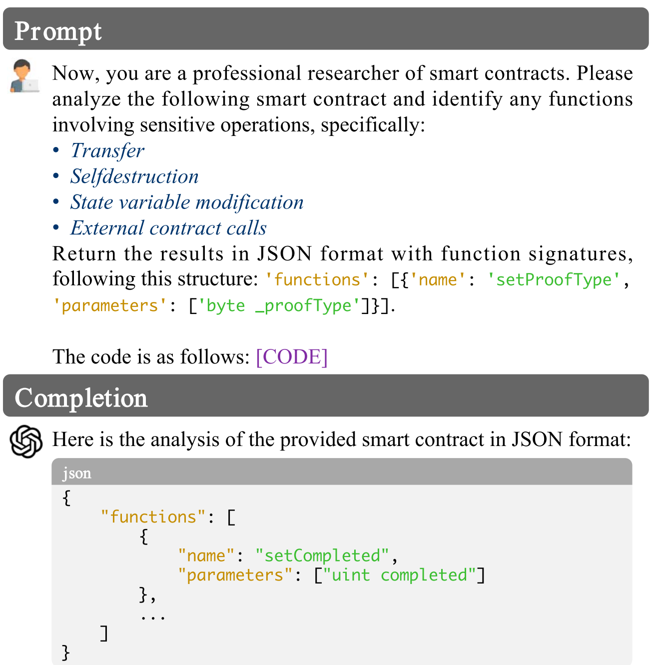

Fig. 4. Prompt template of sensitive function location.

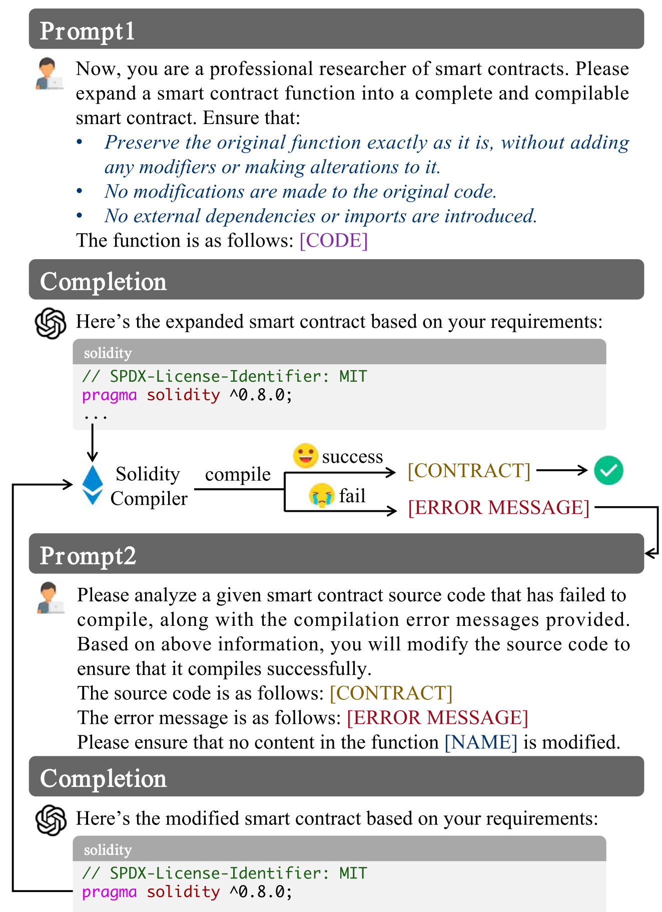

Fig. 5. Prompt template of function snippet completion.

contracts. While acknowledging potential complexities from inter-procedural dependencies within the original repository, providing full (and often fragmented or non-compilable) context to the LLM can be counterproductive due to token limits and the risk of model confusion with noisy inputs. To mitigate these issues, we provide the LLM with only the function's raw source code. The LLM then leverages its semantic understanding to infer a minimal and plausible execution context, generating a self-contained, compilable contract. Although this reconstructed contract serves as a localized approximation of the original environment, it critically enables targeted static analysis of the sensitive function's intrinsic logic, which is often intractable in its initially non-compilable state.

We employ the *gpt-4o* model, guided by the $Prompt1$ template (Fig. 5 [17]), to make the sensitive function snippet "[CODE]" compilable. To ensure the output accurately reflects the original code's security properties, we impose several key constraints in the prompt. Specifically, the LLM is instructed to refrain from altering the snippet or injecting new logic, as such changes could mask existing vulnerabilities or introduce new ones. This approach produces a self-contained contract that preserves the security context of the original function.

Some code snippets compile successfully on the first attempt, while others encounter compilation errors. To address this, we designed a self-reflection mechanism that enables the LLM to modify the completed code until it compiles successfully iteratively. The prompt template for this process, referred to as $Prompt2$, is shown in Fig. 5. Specifically, we provide the

LLM with the completed contract and the associated compilation error messages, guiding it in fixing the code while ensuring the sensitive function snippets in the source code remain unchanged. In $Prompt2$, "[CONTRACT]" represents the completed smart contract, "[ERROR MESSAGE]" denotes the compilation error, and "[NAME]" indicates the sensitive function's name. After receiving the revised smart contract, we attempt compilation again. If the compilation still fails, the self-reflection process continues as described above.

To maintain semantic equivalence during self-reflection, we enforce strict constraints in $Prompt1$ that explicitly prohibit modifications to the original function code. After each iteration, TRACE verifies that the original code snippet remains unchanged by performing string matching after preprocessing. If any modification is detected, the completion process restarts. To prevent infinite loops, we limit self-reflection to a maximum of *five* iterations. If compilation still fails after five attempts, TRACE reports the failure to the user.

C. Vulnerability Detection

The vulnerability detection process is performed individually for each sensitive function identified in the previous steps. That is, each sensitive function snippet is completed into a compilable smart contract, and the following analysis is conducted separately on each of these completed contracts.

1) Construction of FCG: We utilize the *Solidity Compiler* [12] to generate the contract's abstract syntax tree (AST). By analyzing each function node in the AST, we can identify the call relationships for each function. Additionally, modifiers depend on, and visibility attributes associated with each function node are recorded. This step allows us to construct a base function call graph (FCG), where each node includes only the function name, type, and visibility.

Next, we use *Slither* [8] to analyze the contract and generate the function-level control flow graph (CFG), represented in SlithIR. These CFGs are then added to their respective function nodes in the FCG. Furthermore, based on the results from the sensitive function localization step, we annotate each function node with a sensitivity label indicating whether it is a sensitive function.

Finally, the enriched FCG captures all function-call relationships and incorporates five types of detailed information for each function node: the function name, type, visibility, sensitivity label, and function-level CFG.

It is important to note that function snippet completion and vulnerability detection operate at different granularities. During completion, each sensitive function is independently transformed into a standalone contract to enable compilation. However, during vulnerability detection, TRACE reconstructs the inter-procedural relationships by treating each function as a node in the FCG. When analyzing a sensitive function node, TRACE traverses edges to all functions it invokes and its modifiers, extracting their CFGs from the completed contracts.

2) Access Control Localization: Sensitive functions are the primary focus of our analysis. Accordingly, we concentrate on the sensitive function nodes within the FCG. Our analysis begins with an access control localization. The execution permissions of functions are typically enforced by verifying the identity of the contract caller, represented in *Solidity* by the special variable $msg.sender$. To identify such permission checks, we search for logical conditions involving $msg.sender$ (e.g., $TMP_1(boolean) = msg.sender == TMP_0$) within the function's CFG. However, some functions may retrieve the value of $msg.sender$ indirectly, either by calling other functions or through logical checks that depend on its value. To address all possible scenarios, we perform data flow analysis on both operands of all logical operators. Suppose either operand is found to be equal to or dependent on $msg.sender$, we consider the function to include a permission check.

However, permission checks are not always performed in the function itself. In many cases, these checks are implemented in the function's modifier or through specialized permission verification functions. To ensure comprehensive coverage, we evaluate three components in our access control search: the **function itself**, any **functions it invokes**, and its **modifiers**. The CFGs for these components are derived directly from the constructed FCG. If a permission check is identified in any of these components, the function is considered to have proper access control.

3) Risky Actions Identification: Sensitive functions involve operations that, while potentially leading to vulnerabilities (as shown in Table I), can sometimes be executed without access control. For example, deposit and withdraw functions may allow anyone to modify state variables and transfer directly. Risky actions represent a refined subset that exhibits patterns strongly associated with vulnerabilities, based on empirical CVE analysis and security guidelines. We selected four risky action patterns that collectively cover the most common access control vulnerability scenarios. This selection balances comprehensive vulnerability coverage with practical false positive reduction.

Risky Transfer. In smart contracts, fund transfers should typically be accompanied by state modifications to ensure data consistency. For example, a common pattern involves updating a state variable, such as an account balance, before an external call executes the actual transfer. This sequence reflects a standard, safe process. However, a function performing transfers without any associated state variable modifications is considered potentially risky. Fig. 6 illustrates our process for identifying such risky actions within a contract's FCG. We begin by examining the sensitive function's CFG for *transfer* operations. If a *transfer* is detected, we then analyze the function and all its callee functions (traversed via the FCG) to ascertain if any state variables (extracted using *Slither* [8]) are modified. If a transfer occurs without any corresponding state variable modification within this scope, the function is classified as exhibiting a *Risky Transfer*. Conversely, if both a *transfer* and a state variable modification are found along the execution path, no *Risky Transfer* is flagged.

Risky State Variable Modification. Similar to *Risky Transfer*, functions that only modify state variables without accompanying transfers can also pose risks, as they may disrupt the normal functioning of the entire contract. For instance, in

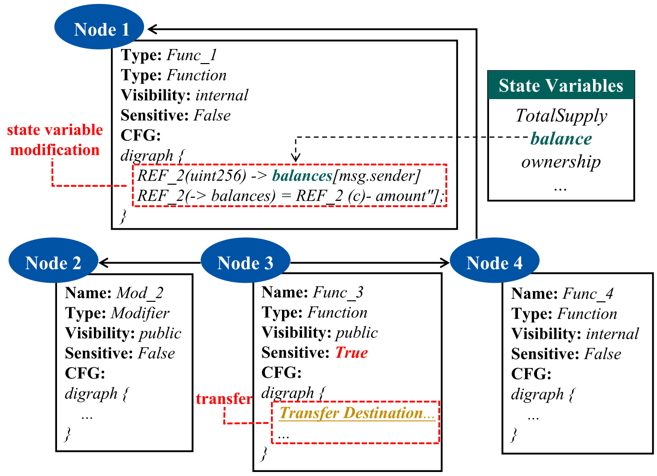

Fig. 6. The process of searching for *risky transfer* in FCG.

the $withdraw$ function from Code 1, if only the balance is updated without executing a transfer, the user's balance could decrease without any actual transaction taking place. The detection method for this type of risk is analogous to the approach used for identifying *Risky Transfer*. We begin by searching for operations that modify state variables within the function. Subsequently, we analyze the functions along the call chain in the FCG to identify any transfer operations. If state variable modifications are present without any corresponding transfer operations, the function is deemed to exhibit *Risky State Variable Modification*.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract SimpleBank {
    mapping(address => uint256) private balances;
    function withdraw(uint256 amount) external {
        require(balances[msg.sender] >= amount, "
            Insufficient balance");
        balances[msg.sender] -= amount;
        // No transfer
    }
}
```

Code 1. A smart contract where *Risky State Variable Modification* exists.

Low-level External Contract Call. External calls in smart contracts can be made using various methods, including low-level, intermediate, and high-level calls. Low-level calls refer to direct calls to other contracts using the call function, which allows invoking any function at any address and requires the caller to handle the results and errors manually. Among these, low-level calls are the riskiest, as they bypass *Solidity*'s type-checking and error-handling mechanisms. This exposes the contract to potential attacks, where an attacker could design a malicious contract and use a low-level call to trigger its harmful logic within the victim contract. Slither's intermediate representation (SlithIR) [31] clearly distinguishes between different function call methods, with low-level calls denoted as LOW_LEVEL_CALL. Accordingly, we search the function's CFG for this representation. If it is identified, we classify the function as exhibiting *Low-level External Contract Call*.

Selfdestruct. *Selfdestruct* is a high-risk operation. Once executed, this function irreversibly removes the contract from the blockchain and transfers all remaining balances to a designated address. As a result, it is crucial to implement strict access control mechanisms to prevent unauthorized execution. In SlithIR, the *Selfdestruct* function is represented as $SOLIDITY\_CALL\ selfdestruct$. To identify its presence, we directly search for this representation within the CFG.

While our rule design emphasizes broad coverage of risky patterns, it inevitably introduces the potential for over-approximation in certain scenarios. For example, the *Risky State Variable Modification* rule considers any state-changing operation without an accompanying transfer as potentially risky. However, this assumption may not hold universally—some contracts implement legitimate business logic that updates state without involving any token or value transfer. Recognizing this, TRACE is intentionally designed to report risks rather than confirmed vulnerabilities, offering developers guidance on potentially security-relevant behaviors while leaving final validation to human auditors.

4) Detection Algorithm: The final detection step systematically analyzes the completed smart contract to identify Access Control vulnerabilities. The logic, formalized in Algorithm 1, is designed to determine whether sensitive functions contain inadequately protected risky actions. The algorithm takes a completed smart contract as input and returns a list of detected vulnerabilities, each represented as a tuple containing the vulnerable function and the specific location of the unprotected risky action. The detection proceeds in six steps: (1) constructing an enriched FCG with CFG annotations for each function node; (2) iterating through all function nodes in the FCG; (3) for each sensitive function, searching for access control checks in the function itself, its modifiers, and functions it invokes; (4) identifying all risky actions within the sensitive function using the patterns defined in Section III-C3; (5) comparing the locations of access control checks and risky actions, if a risky action occurs before any access control check or if no check exists, it is flagged as a vulnerability; (6) returning all detected vulnerabilities for manual review.

IV. EVALUATION

To evaluate the performance of TRACE, we designed four research questions (RQs) to assess its effectiveness across different stages and scenarios. The RQs are as follows:

RQ1. How effective is TRACE in *Sensitive Function Extraction*?

RQ2. How effective is TRACE in *Function Snippet Completion*?

RQ3. How does TRACE perform in detecting single smart contracts deployed on blockchain?

RQ4. How effective is TRACE in detecting real-world smart contract repositories?

A. Experimental Setup

1) Datasets: We utilized three datasets for evaluation. The first dataset comprises **vulnerable smart contracts**, including

Algorithm 1 Detection of Access Control Vulnerabilities

Input: A smart contract *SC*
Output: Detected vulnerabilities *VulnList*

```
FCG ← ConstructFCG(SC)
ACLocation ← null
RALocations ← ∅
VulnList ← ∅
for function f ∈ FCG.nodes do
    if FCG.getLabel(f) ≠ sensitive then
        continue
    end if
    ACLocation ← AccessControlSearch(f)
    RALocations ← RiskyActionsSearch(f)
    if RALocations ≠ ∅ then
        for location r ∈ RALocations do
            if ACLocation = ∅ or ACLocation > r then
                Add (f, r) to VulnList
            end if
        end for
    end if
end for
return VulnList
```

15 contracts that have been assigned CVEs [32]. These contracts predominantly exhibit typical *Access Control* vulnerabilities and are older versions predating *Solidity* 0.6.0. Notably, this dataset has also been used in prior evaluations of AChecker [19] and SPCon [20]. The second dataset contains 5,000 recent smart contracts, all of which have been **deployed on-chain** and verified with validated source code. These contracts were collected from *Etherscan* [33], with the latest date being November 12, 2024. The third dataset focuses on **smart contract repositories** obtained from *DAppSCAN* [24], which is a large-scale SWC weakness [34] dataset from real-world DApps. Using a confidence level of 95% and a confidence interval of 10 [35], we sampled the *DAppSCAN* dataset and randomly selected 83 smart contract repositories, encompassing a total of 3,092 contracts.

We evaluate TRACE using three datasets, each designed to address specific research questions. Dataset 1 (CVE contracts) and Dataset 2 (on-chain contracts) are used for RQ3 to evaluate detection performance on compilable contracts with known vulnerabilities and large-scale real-world deployments, respectively. Dataset 3 (repositories) is used for RQ4 to evaluate TRACE's capability on non-compilable repositories. Additionally, RQ1 and RQ2 use sampled subsets from Dataset 3 to assess sensitive function extraction and snippet completion effectiveness.

2) Setup: This section details the experimental setup, including the environment and models used to evaluate TRACE's performance.

Experimental environment. Our experiments were conducted on a machine running Ubuntu 22.04.5 LTS, equipped with 10 CPU cores and 20 GB of RAM. To ensure consistency and efficiency, we imposed a time limit of 30 minutes for each tool to analyze a single smart contract.

Model. Apart from using other models for the comparative experiments, we choose to use *gpt-4o* for all the tasks. All the models were accessed via API calls. We use the *gpt-4o* model API with a maximum context window of 128,000 tokens, which accommodates the vast majority of function snippets and completed contracts in our datasets. Temperature is set to 0.7, and max new tokens for generation is set to 4,096. For edge cases where code snippets exceed the token limit, TRACE reports a processing failure.

3) Method: The evaluation dataset comprises both single smart contracts deployed on blockchain and non-compilable smart contract repositories. To accommodate this distinction, we designed two experimental approaches. For single smart contracts deployed on blockchain, all contracts in the dataset are compilable, eliminating the need to address potential compilation failures. In this case, we bypassed intervention with the LLMs by skipping the *Sensitive Function Extraction* and *Function Snippet Completion* steps. Instead, we analyzed every function within the contract as if it were a sensitive function. While this approach may increase the computational cost, it ensures more comprehensive analysis results. For non-compilable smart contract repositories, we applied the methodology detailed in Section III to perform the analysis.

B. Results

In this section, we present the experimental results addressing the research questions formulated in Section IV.

1) RQ1: From the 3,092 contracts in the repository dataset, we filtered out secure contracts, such as libraries and interfaces (see Section III-A), leaving 1,781 contracts that potentially contain sensitive functions. Using a confidence level of 95% and a confidence interval of 10, we randomly sampled a subset of 91 contracts from this dataset, following the experimental setup outlined in related research [36]. All sensitive functions in the 91 contracts were manually labeled. Specifically, two researchers, each with over two years of experience in smart contract security auditing, independently labeled the sensitive functions. Any disagreements were resolved through discussion to reach a final consensus. This process ensures the quality and consistency of the labels used for evaluation. Subsequently, we tasked the LLM with extracting sensitive functions from the subset. We calculate precision using formula $P = TP/(TP + FP)$, recall using formula $R = TP/(TP + FN)$, F1 using formula $F1 = 2 * P * R/(P + R)$.

Among all the functions analyzed across 91 sampled contracts, the LLM identified 325 true positives (TP), 3 false positives (FP), 913 true negatives (TN), and 10 false negatives (FN). As a result, the LLM achieved an accuracy of 99.1%, a recall of 97.0%, and an F1 score of 98.0% in the task of sensitive function extraction, demonstrating strong performance. Upon analysis of the three false positives, we found that all were caused by hallucinations, where the LLM generated a function signature that did not exist in the original code. Conversely, the majority of the ten false negatives resulted from the extensive length of certain sensitive functions, which negatively impacted the model's inference performance.

Answer to RQ1: TRACE demonstrated excellent performance in sensitive function extraction, achieving an F1 score of *98.0%*.

2) RQ2: In the repository dataset, there are 83 repositories comprising a total of 3,092 contract files. From these files, we extracted 4,263 sensitive functions, each of which was used to generate a completed smart contract, resulting in 4,263 completed contracts. To evaluate TRACE's capability in function snippet completion, we employed two metrics: 1) **Number of contracts that are uncompilable**: Our objective is to compile contracts containing sensitive functions and subsequently conduct static analysis. Therefore, the ability of the completed contracts to successfully compile is a critical criterion. We used the *Solidity* compiler to compile these contracts and recorded the number of failures. 2) **Number of contracts where the original code was modified**: In addition to ensuring successful compilation, it is vital that the original sensitive function snippets remain unaltered during the completion process. Modifications to the original code could introduce new vulnerabilities or inadvertently resolve existing ones, potentially affecting the accuracy of the final detection results. To determine whether the original code was modified, we preprocessed the code before and after completion by removing irrelevant characters (e.g., extra spaces, line breaks, comments). We then attempted to locate the original code in the completed contract. If the search failed, we considered the original code to have been modified, and any subsequent results will be disregarded.

According to the experimental results, out of a total of 4,263 contracts, 4,169 were successfully compiled after the process of completion and self-reflection, achieving a success rate of 97.8%. Furthermore, 4,051 contracts remained unmodified, accounting for 95.0% of the total. These results demonstrate that the LLM performs effectively in function snippet completion.

Answer to RQ2: TRACE achieved a 97.8% compilation success rate and ensured 95.0% of contracts being completed correctly, demonstrating strong performance in function snippet completion.

3) RQ3: We began by evaluating TRACE using a dataset containing known vulnerabilities. Following the experimental framework outlined by *AChecker* [19], we included a selection of tools integrated with smartbugs [37] (i.e., *Slither* [8], *Maian* [23], *Manticore* [60], *SmartCheck* [10], and *Mythril* [22]), each capable of detecting at least one type of *Access Control* vulnerability. Additionally, *SPCon* [20] and *GPTScan* were incorporated into this comparative analysis. We also included *AChecker* [19] itself as a baseline for comparison. Certain tools were excluded from our analysis because they failed to identify any vulnerabilities, reporting no positive results. The experimental results for all tools on this dataset are presented in Table II.

TRACE successfully detected 14 out of 15 CVEs, achieving a recall rate of 93%. The one missed vulnerability, CVE-2020-35962, could not be detected due to compilation issues; notably, no other tool was able to identify this vulnerability either. Among the comparative tools, *AChecker* performed the best, identifying 12 CVEs and achieving a recall rate of 80%. These results underscore the superior comprehensiveness of TRACE in detecting vulnerabilities in smart contracts.

TABLE II
COMPARISON WITH OTHER SOTA TOOLS ON THE VULNERABLE DATASET

CVE	**Slither**	**Maian**	**SmartCheck**	**Mythril**	**SPCon**	**AChecker**	**Ours**
CVE-2018-10666					✓	✓	✓
CVE-2018-10705					✓	✓	✓
CVE-2018-11329					✓	✓	✓
CVE-2018-19830						✓	✓
CVE-2018-19831				✓	✓	✓	✓
CVE-2018-19832		✓		✓	✓	✓	✓
CVE-2018-19833						✓	✓
CVE-2018-19834						✓	✓
CVE-2019-15078		✓		✓	✓	✓	✓
CVE-2019-15079					✓		✓
CVE-2019-15080					✓	✓	✓
CVE-2020-17753	✓		✓	✓			✓
CVE-2020-35962							
CVE-2021-34272					✓	✓	✓
CVE-2021-34273						✓	✓
Recall%	6	13	6	26	60	80	**93**

Code 2 presents a simplified version of CVE-2019-15079, an access control vulnerability that evaded detection by *AChecker* but was successfully identified by TRACE. The vulnerability in this contract arises from a **misnamed constructor**, i.e., `EAI_TokenERC` was misnamed as `EAI_TokenERC20`, making a sensitive initialization function publicly accessible and callable by any user. TRACE detects this issue by leveraging an LLM to recognize the function's privileged semantics (e.g., state variable modification) and identifies the lack of access control through static analysis. In contrast, *AChecker* missed this vulnerability because it failed to recognize the function as sensitive, due to its reliance on predefined patterns for critical instructions. Moreover, it lacks semantic reasoning capabilities to detect misnamed constructors or to infer intended behavior from code context. As a result, it cannot flag the public accessibility of a privileged initialization function as a security risk.

```
pragma solidity ^0.4.16;
contract EAI_TokenERC {
    uint256 public totalSupply;
    mapping (address => uint256) public balanceOf;
    function EAI_TokenERC20(
        uint256 initialSupply,
        string tokenName,
        string tokenSymbol
    ) public {
        totalSupply = initialSupply * 10 ** 8;
        balanceOf[msg.sender] = totalSupply;
    }
}
```

Code 2. Vulnerabilities that cannot be detected by AChecker but can be detected by Trace.

Since the contracts in the CVE dataset all contain vulnerabilities and the size of the dataset is limited, we also validate the effectiveness of TRACE on a larger-scale real-world dataset (on-chain dataset). For the experiments, we selected the same tools used

TABLE III
COMPARISON WITH SOTA TOOLS ON THE ON-CHAIN DATASET

Tools	# TP	# FP	# Failure	Precision%
Manticore	0	0	322	0
Maian	0	0	1,447	0
Mythril	1	2	601	33.3
AChecker	4	7	485	36.3
Slither	90	71	35	55.9
GPTScan	80	24	76	76.9
TRACE	91	11	58	**89.2**

for comparison. Notably, some tools are limited to accurately analyzing early versions of *Solidity* [12] (e.g., versions prior to v0.6.0), whereas most contracts in the on-chain dataset are written using newer *Solidity* versions (e.g., v0.8.0 and later). As a result, tools like *Oyente* [9] and *SmartCheck* [10] were excluded from the comparison. In contrast, TRACE includes all *Solidity* compiler versions up to 0.8.30 (the latest at the time of our experiments) and can continuously integrate new versions, ensuring broad compatibility. Furthermore, *SPCon* [20], which relies on the transaction history of smart contracts for analysis, could not be applied because the latest on-chain contracts have insufficient transaction records to meet its analysis requirements. The final experimental results are presented in Table III.

We recorded all positive results reported by the tools and manually verified each one. Each positive result was manually verified by two smart contract researchers, each with over two years of experience. To assess inter-rater reliability, we calculated Cohen's Kappa coefficient [38] on the 102 initially flagged cases: 96 cases showed agreement (both marked as vulnerable or secure), while 6 cases showed disagreement. This yielded k = 0.88, indicating substantial agreement. Discrepancies in their findings were resolved through discussion until a consensus was reached. This entire verification process concluded within two weeks. The analysis revealed that TRACE detected 91 true positives and 11 false positives, achieving a precision of 89.2%. This result makes TRACE the most precise tool, outperforming the second-ranked *GPTScan* [18] by a margin of 12.3%. Robustness is another critical factor in evaluating tool performance, as some tools encounter timeouts when analyzing highly complex contracts. Thus, beyond positive results, we also considered the number of contracts that each tool failed to analyze. As shown in Table III, *Slither* had the lowest number of failed analyses, with 35 contracts, while TRACE reported slightly more failures, at 58. This is partly due to TRACE 's approach of analyzing each function independently when evaluating individual contracts (see Section IV-A3), which increases computational costs. However, as the results demonstrate, this trade-off remains acceptable, given the tool's superior detection performance. The high failure rates for some tools, such as *Maian* [23] and *Manticore* [60], stem from two factors. First, these tools employ dynamic analysis methods, which are inherently more time-consuming than static analysis. Second, these tools may encounter compilation issues due to their own design constraints when processing contracts with complex dependencies or newer *Solidity* features.

Answer to RQ3: TRACE demonstrated superior detection performance, achieving a 93% recall on CVEs and 89.2% precision on a large-scale on-chain dataset, outperforming comparative SOTA tools.

4) RQ4: We compared TRACE with a baseline LLM for detecting vulnerabilities in non-compilable repositories. Recognizing that strict output formatting constraints can negatively affect LLM performance and increase hallucination rates—as observed in prior work [16]—we followed their recommendation. A two-stage prompting strategy was adopted to mitigate this issue.

As illustrated in Fig. 7, the first stage instructs the LLM to act as a smart contract security expert and perform unconstrained vulnerability analysis on a given contract. It receives as input the source code along with a description of the four types of access control vulnerabilities. It outputs a natural-language analysis that either (1) lists the type(s) of vulnerabilities detected and the name(s) of the affected functions, or (2) states that no vulnerabilities were found. In the second stage, a new LLM session is initiated to semantically parse the output from the first stage and convert it into a structured format compatible with downstream processing. This separation between reasoning and formatting improves reliability by avoiding complex prompt constraints during analysis, while still producing consistent outputs for evaluation.

We selected five of the latest and widely used models for comparison: *Gemini-2-flash-exp* [39], *Claude-3.5-sonnet* [40], *o1* [41], *GPT-4o* [42] and *DeepSeek-R1* [43]. The experimental results are summarized in Table IV. Among these models, *DeepSeek-R1* achieved the highest precision, detecting 26 true positives (TPs) with a precision of 14.3%. However, it also produced a significant number of false positives, totaling 156. This aligns with findings from prior research [16], which highlight that LLMs often generate a high volume of false positives during vulnerability detection tasks, leading to low overall precision.

In contrast, TRACE does not rely on LLMs for direct vulnerability detection. Instead, it utilizes LLMs exclusively for code understanding and completion. The actual vulnerability detection process is carried out through static analysis, effectively mitigating the issue of excessive false positives associated with LLM-based detection. As a result, TRACE identified 40 TPs with a substantially higher precision of 87.0%. Nevertheless, LLMs demonstrate superior robustness. All five models successfully processed the contracts without encountering detection failures, thanks to their ability to support extended input lengths. In contrast, TRACE failed to analyze five contracts due to the complexity of the completed contracts, leading to analysis timeouts. However, given TRACE 's significantly higher precision, we consider these occasional timeouts to be an acceptable trade-off.

Our analysis of TRACE's six false positives showed that four were caused by the LLM component, which introduced new vulnerabilities by modifying secure code. The other two were a direct result of a design trade-off: to manage computational costs, our static analyzer is limited to a function call depth of

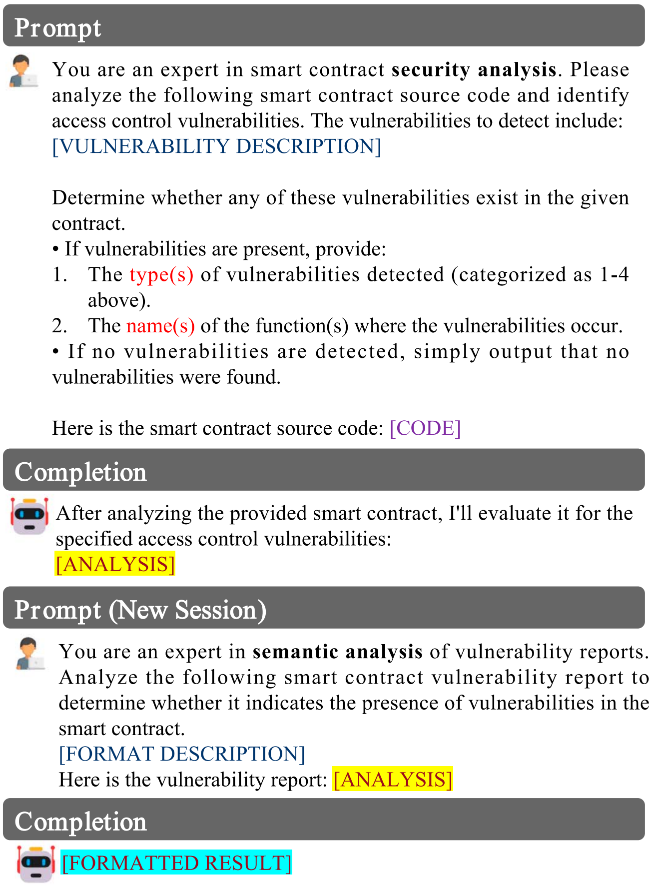

Fig. 7. Two-stage prompt template for vulnerability detection using LLMs. The first prompt performs free-form analysis on the contract source, while the second prompt parses and formats the analysis result.

TABLE IV
COMPARISON WITH LLMS ON THE REPOSITORY DATASET

LLMs	# TP	# FP	# Failure	Precision%
Gemini-2-flash-exp	34	547	0	5.6
Claude-3.5-sonnet	28	387	0	6.7
o1	14	108	0	11.5
GPT-4o	22	141	0	13.5
DeepSeek-R1	26	156	0	14.3
TRACE	40	6	5	**87.0**

three. Consequently, it failed to detect access control statements located in call chains deeper than this pre-set limit. For large-scale datasets where comprehensive recall calculation is impractical, we analyze potential causes of missed vulnerabilities. False negatives can arise from two primary sources similar to false positives. First, LLM timeouts when processing contracts with highly complex contexts or during iterative self-reflection can prevent complete analysis. Second, some sensitive operations may be executed through call chains exceeding our three-level depth limit. In such cases, risky actions deep in call chains may not be identified, leading to improperly protected functions being incorrectly classified as secure.

> **Answer to RQ4:** TRACE achieved a precision of 87.0% on smart contract repositories, outperforming five popular LLMs.

5) Cost Analysis: We analyze the monetary cost of using TRACE, which relies on *gpt-4o* API calls for sensitive function extraction and completion. For the on-chain dataset (5,000 contracts), TRACE consumed approximately \$127 in total API costs, averaging \$0.025 per contract. For the repository dataset (83 repositories containing 3,092 contracts), the total cost was approximately \$124, averaging \$1.49 per repository or \$0.04 per contract. These costs are reasonable for security auditing scenarios where comprehensive vulnerability detection can prevent significant financial losses.

V. DISCUSSION

A. Implications

For Third-Party Developers. TRACE directly addresses the pain points of code reuse and security validation in decentralized development. By removing the need for compilation, the tool broadens access to security analysis, enabling developers to efficiently analyze complex, dependency-laden repositories.
For Smart Contract Repository Development Teams. While development teams typically operate within compilable environments, TRACE offers a competitive advantage in security assurance. Its superior detection performance compared to SOTA tools (see Section IV) suggests that even teams with mature workflows could benefit from integrating TRACE into their security pipelines.
For Researchers. Our approach allows for large-scale analysis of historical repositories that are non-compilable due to deprecated dependencies or toolchain changes, enabling in-depth exploration of their security aspects. Moreover, the tool's architecture is extendable, allowing for the integration of additional detection heuristics and the incorporation of new vulnerability types, thereby supporting ongoing advancements in smart contract security research.

B. Complexity of Compiling Repositories

We performed local compilation on the repository-level dataset, which contains 83 smart contract repositories. This included attempts to compile both single smart contract files in the repository and the entire repository. Table V presents key complexity metrics related to the dataset. At the file level, we used the exact solc version specified in each of the 3,092 contracts. Widespread dependency and compatibility issues resulted in a low compilation success rate of only 30.6%.

At the repository level, we used Truffle [26][1] to compile all 83 repositories. Compilation was further complicated by dependencies on 13 external libraries, four of which lack *Truffle* support. This led to an even lower success rate of 6.0% (5/83), highlighting that the vast majority of repositories cannot be compiled without significant manual intervention.

[1] *Truffle* has been officially deprecated. We use it here for historical consistency with some older repositories in the dataset.

TABLE V
COMPLEXITY OF REPOSITORY DATASET

Metric	Level	Value
Average lines of code	Contract	139.4
Internal file dependencies	Contract	6,792
Compilation success rate	Contract	945 / 3,092 (30.6%)
Average number of contracts	Repository	37.3
External library dependencies	Repository	13 (4 unsupported)
Compilation success rate	Repository	5 / 83 (6.0%)

C. Threats to Validity

Internal Threats. The use of LLMs introduces inherent time and computational costs. However, this overhead is justified by the ability of LLMs to fully automate the analysis of smart contract repositories – an advantage over traditional tools that often require manual effort to ensure successful compilation. While this work focuses specifically on access control vulnerabilities, the underlying framework of TRACE is extensible. It can be adapted to support the detection of additional vulnerability types by extending the set of sensitive operations and analysis rules accordingly.

External Threats. During the sensitive function extraction and function snippet completion stages, TRACE leverages LLMs, which are prone to inherent limitations such as hallucination and data leakage. To address this, we validate the results generated by the LLMs. If the results contain non-existent function signatures or alter the original code, TRACE will detect and report these discrepancies.

VI. RELATED WORK

A. Large Language Models

Large Language Models (LLMs) have been widely applied in various tasks in the field of software engineering. Ahmed et al. [44] investigate the use of few-shot training with the GPT Codex model and demonstrate that project-specific training can significantly outperform state-of-the-art models for code summarization. Wang et al. [15] propose *RLCoder*, a novel reinforcement learning framework for repository-level code completion that enables unsupervised learning of useful content retrieval. Chen et al. [16] and David et al. [45] studied the performance of LLMs in detecting smart contract vulnerabilities and identified their low precision. Unlike these approaches, we do not directly employ LLMs for vulnerability detection. Instead, we leverage them for code understanding and completion to support program analysis. Kim et al. [46] explore the use of a lightweight transformer-based NLP model to detect vulnerabilities in smart contracts. Ma et al. [47] present *iAudit*, a multi-agent system for detecting and explaining Solidity smart-contract vulnerabilities.

Recent research has demonstrated the potential of LLMs in various program verification tasks beyond vulnerability detection. Pei et al. [48] investigated whether LLMs can generate program invariants, demonstrating effectiveness on synthetic Java programs. Wu et al. [49] proposed LEMUR, which integrates LLMs with symbolic verifiers through a proof calculus for loop invariant synthesis. Misu et al. [50] developed methods for AI-assisted synthesis of verified Dafny methods, while Yang et al. [51] presented AutoVerus for automated proof generation in Rust code using the Verus verifier. Kamath et al. [52] explored LLM-driven loop invariant generation for C programs using the Frama-C verification framework. However, these approaches primarily focus on generating proofs or invariants for already compilable code in established verification frameworks. TRACE fundamentally differs by using LLMs to enable analysis of non-compilable repositories, a preprocessing step that makes program analysis feasible where it was previously impossible.

B. Smart Contract Vulnerability Detection

Smart contract vulnerability detection has also received widespread attention from researchers. Static analysis is a widely used method for detecting vulnerabilities in smart contracts, as it examines the source code without executing it, allowing for the identification of potential issues early in the development process [53]. Luu et al. [54] develop a symbolic execution tool, *Oyente*, which identifies vulnerabilities in existing contracts. Chen et al. [55] identify 20 types of defects in smart contracts, and provide a dataset to help developers prioritize defect removal for improved contract quality. Feist et al. [8] present *Slither*, a static analysis framework for Ethereum smart contracts that converts *Solidity* contracts into an intermediate representation for efficient bug detection, optimization, and code review. Tikhomirov et al. [56] provide a comprehensive classification of Solidity code issues and present *SmartCheck*, an extensible static analysis tool for detecting vulnerabilities in smart contracts. Bose et al. [57] present *Sailfish*, a scalable system for detecting state-inconsistency bugs in smart contracts, combining a lightweight exploration phase with precise symbolic evaluation guided by value-summary analysis. Dynamic analysis is also an effective solution for vulnerability detection [58]. Mossberg et al. introduce *Manticore* [59], a dynamic symbolic execution framework for analyzing binaries and Ethereum smart contracts. Grieco et al. [60] introduce *Echidna*, a smart contract fuzzer designed for automatic test generation to detect property violations. *AChecker* [19] is a static analysis tool designed to detect access control vulnerabilities via data-flow analysis. While effective on compilable contracts, *AChecker* cannot handle non-compilable repositories as it depends on generating complete intermediate representations from compiled code. In contrast, TRACE leverages LLMs to overcome this limitation, enabling vulnerability detection in non-compilable smart contract repositories. Moreover, TRACE employs a more refined two-tier classification and function-level analysis strategy, enabling more precise vulnerability identification compared to *AChecker*'s broader data-flow approach.

VII. CONCLUSION

We introduced TRACE, a novel tool for detecting vulnerabilities in non-compilable smart contract repositories by combining LLM capabilities with traditional static analysis. Evaluations show TRACE robustly outperforms existing tools: it achieved 89.2% precision on an on-chain dataset, exceeding *GPTScan* by 12.3%, and 87.0% precision on a repository dataset, significantly surpassing *DeepSeek-R1* (14.3%). This work underscores the promise of combining LLMs with program analysis for intricate smart contract security challenges.

References

[1] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "DefectChecker: Automated smart contract defect detection by analyzing EVM bytecode," *IEEE Trans. Softw. Eng.*, vol. 48, no. 7, pp. 2189–2207, Jul. 2022.

[2] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. Leung, "Decentralized applications: The blockchain-empowered software system," *IEEE Access*, vol. 6, pp. 53019–53033, 2018.

[3] S. Chitta, S. M. Yellepeddi, S. Thota, and A. K. P. Venkata, "Decentralized finance (DeFi): A comprehensive study of protocols and applications," *Distrib. Learn. Broad Appl. Scientific Res.*, vol. 5, pp. 124–145, Jan. 2019.

[4] M. Müller, S. R. Garzon, M. Westerkamp, and Z. A. Lux, "HIDALS: A hybrid IoT-based decentralized application for logistics and supply chain management," in *Proc. IEEE 10th Annu. Inf. Technol., Electron. Mobile Commun. Conf. (IEMCON)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 0802–0808.

[5] A. Satybaldy, A. Hasselgren, and M. Nowostawski, "Decentralized identity management for e-health applications: State-of-the-art and guidance for future work," *Blockchain Healthc. Today*, vol. 5, no. S1, 2022.

[6] "Vulnerability: Improper access control," OWASP, 2023. [Online]. Available: https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC04-access-control-vulnerabilities.html

[7] L. Breidenbach, P. Daian, A. Juels, and E. G. Sirer, "An in-depth look at the parity multisig bug," 2017. [Online]. Available: https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/

[8] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 8–15.

[9] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.

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

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

[12] "Solidity," 2024. [Online]. Available: https://soliditylang.org

[13] L. Guo et al., "When to stop? towards efficient code generation in LLMs with excess token prevention," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2024, pp. 1073–1085, doi: 10.1145/3650212.3680343.

[14] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, 2024, pp. 1–13.

[15] Y. Wang et al., "RLCoder: Reinforcement learning for repository-level code completion," 2024. [Online]. Available: https://arxiv.org/abs/2407.19487

[16] C. Chen et al., "When ChatGPT meets smart contract vulnerability detection: How far are we?" *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 4, 2023.

[17] J. Chen et al., "Identifying smart contract security issues in code snippets from stack overflow," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2024, pp. 1198–1210.

[18] Y. Sun et al., "GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2024, doi: 10.1145/3597503.3639117.

[19] A. Ghaleb, J. Rubin, and K. Pattabiraman, "AChecker: Statically detecting smart contract access control vulnerabilities," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 945–956.

[20] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2022, pp. 716–727.

[21] "Ethereum," 2024. [Online]. Available: https://ethereum.org/en/

[22] ConsenSys. "Mythril: Security analysis tool for EVM bytecode." GitHub. Accessed: Apr. 20, 2025. [Online]. Available: https://github.com/ConsenSys/mythril

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

[24] Z. Zheng, J. Su, J. Chen, D. Lo, Z. Zhong, and M. Ye, "DAppSCAN: Building large-scale datasets for smart contract weaknesses in DAPP projects," *IEEE Trans. Softw. Eng.*, vol. 50, pp. 1360–1373, 2024.

[25] "Hardhat," 2024. [Online]. Available: https://hardhat.org/

[26] Truffle suite. [Online]. Available: https://archive.trufflesuite.com/

[27] Openzeppelin. Accessed: Apr. 20, 2025. [Online]. Available: https://www.openzeppelin.com

[28] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Softw.*., vol. 25, no. 5, pp. 22–29, Sep.–Oct. 2008.

[29] "Ethereum virtual machine," Wikipedia, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Ethereum#Virtual_machine

[30] "Vyper," 2024. [Online]. Available: https://docs.vyperlang.org/en/stable/

[31] crytic. "Slithir." GitHub. Accessed: Apr. 20, 2025. [Online]. Available: https://github.com/crytic/slither/wiki/SlithIR

[32] "Cve," 2024. [Online]. Available: https://www.cve.org

[33] "Etherscan," 2024. [Online]. Available: https://etherscan.io

[34] "Smart contract weakness classification," 2024. [Online]. Available: https://swcregistry.io/

[35] "Sample size calculator," 2024. [Online]. Available: https://www.surveysystem.com/sscalc.htm

[36] S. Yang, J. Chen, and Z. Zheng, "Definition and detection of defects in NFT smart contracts," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2023, pp. 373–384, doi: 10.1145/3597926.3598063.

[37] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "SmartBugs: A framework to analyze Solidity smart contracts," in *Proc. 35th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2020, pp. 1349–1352.

[38] S. M. Vieira, U. Kaymak, and J. M. Sousa, "Cohen's kappa coefficient as a performance measure for feature selection," in *Proc. Int. Conf. Fuzzy Syst.*, Piscataway, NJ, USA: IEEE Press, 2010, pp. 1–8.

[39] "Gemini 2.0 flash," Google, 2024. [Online]. Available: https://ai.google.dev/gemini-api/docs/models/gemini-v2?hl=zh-cn

[40] "Claude 3.5 sonnet," Anthropic, 2024. [Online]. Available: https://www.anthropic.com/claude/sonnet

[41] "o1," OpenAI, 2024. [Online]. Available: https://openai.com/o1/

[42] "GPT-4o," OpenAI, 2024. [Online]. Available: https://platform.openai.com/docs/models#gpt-4o

[43] D. Guo et al., "DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2501.12948

[44] T. Ahmed and P. Devanbu, "Few-shot training LLMs for project-specific code-summarization," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, New York, NY, USA: ACM, 2023, doi: 10.1145/3551349.3559555.

[45] I. David, L. Zhou, K. Qin, D. Song, L. Cavallaro, and A. Gervais, "Do you still need a manual smart contract audit?" 2023. [Online]. Available: https://arxiv.org/abs/2306.12338

[46] J. Kim, T.-T.-H. Le, S. Lee, and H. Kim, "Ethereum smart contracts vulnerabilities detection leveraging fine-tuning distilBERT," in *Proc. Int. Conf. Platform Technol. Service (PlatCon)*, 2024, pp. 133–138.

[47] W. Ma et al., "Combining fine-tuning and LLM-based agents for intuitive smart contract auditing with justifications," 2024, *arXiv:2403.16073*.

[48] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can large language models reason about program invariants?," in *Proc. Int. Conf. Mach. Learn.*, 2023, pp. 27496–27520.

[49] H. Wu, C. Barrett, and N. Narodytska, "Lemur: Integrating large language models in automated program verification," 2023, *arXiv:2310.04870*.

[50] M. R. H. Misu, C. V. Lopes, I. Ma, and J. Noble, "Towards AI-assisted synthesis of verified Dafny methods," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 812–835, 2024.

[51] C. Yang et al., "AutoVerus: Automated proof generation for rust code," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, pp. 3454–3482, 2025.

[52] A. Kamath et al., "Leveraging LLMs for program verification," in *Proc. Formal Methods Comput.-Aided Des. (FMCAD)*, Piscataway, NJ, USA: IEEE Press, 2024, pp. 107–118.

[53] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security Privacy*, vol. 2, no. 6, pp. 76–79, Nov.–Dec. 2004.

[54] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. (CCS)*, New York, NY, USA: ACM, 2016, pp. 254–269, doi: 10.1145/2976749.2978309.

[55] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on Ethereum," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 327–345, Jan. 2022.

[56] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of Ethereum smart contracts," in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, New York, NY, USA: ACM, 2018, pp. 9–16, doi: 10.1145/3194113.3194115.

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

[58] T. Ball, "The concept of dynamic analysis," in *Proc. 7th Eur. Softw. Eng. Conf. Held Jointly with 7th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (ESEC/FSE-7)*, Berlin, Heidelberg: Springer-Verlag, 1999, pp. 216–234.

[59] M. Mossberg et al., "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2019, pp. 1186–1189.

[60] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, usable, and fast fuzzing for smart contracts," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2020, pp. 557–560, doi: 10.1145/3395363.3404366.