ACTAINT: Agent-Based Taint Analysis for Access Control Vulnerabilities in Smart Contracts

Huarui Lin[1,†], Zhipeng Gao[†], Jiachi Chen[†], Xiang Chen[‡], Xiaohu Yang[†], Lingfeng Bao[1,2,†]

The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China[†]
Nantong University, China[‡]

12321077@zju.edu.cn, zhipeng.gao@zju.edu.cn, chenjiachi317@gmail.com, xchencs@ntu.edu.cn, yangxh@zju.edu.cn, lingfengbao@zju.edu.cn

***Abstract*—Smart contracts have become a foundational component of blockchain systems, enabling decentralized, transparent, and autonomous execution of application logic across various domains, including decentralized finance (DeFi), gaming, and digital identity. Due to their immutable and trustless nature, smart contracts often manage and transfer substantial amounts of assets without human intervention. However, vulnerabilities in smart contracts can lead to substantial financial losses. Among these, access control vulnerabilities are particularly critical, typically originating from inadequately designed or incorrectly implemented permission mechanisms. Most existing methods for detecting access control vulnerabilities are based on static analysis, which heavily relies on manually defined rules and pattern matching. While these methods are efficient at identifying certain classes of known vulnerabilities, they are inherently limited in scope and generalization. In particular, they often fail to capture the underlying business logic of smart contracts.**

In this paper, we propose an LLM-based multi-agent system, named ACTAINT, for detecting access control vulnerabilities in Solidity smart contracts. ACTAINT first performs static analysis to guide the sink agent in identifying potential sinks. Then, based on the identified sinks, the taint agent conducts taint analysis to determine whether a data flow exists from untrusted sources to these sinks. We evaluate our approach on three datasets: known CVE cases, a set of 624 real-world smart contracts, and another set of 93 real-world smart contracts. The results demonstrate that our method outperforms existing tools in both datasets. On the first dataset, our approach outperforms state-of-the-art tools, including AChecker and GPTLens, achieving higher recall and F1-score. On the second dataset, our method surpasses the leading static analysis tool AChecker, with a 8.3% improvement in precision and an 9.7% improvement in F1-score.

I. INTRODUCTION

Smart contracts are Turing-complete programs that can execute any computation a programmable computer can perform. They are one of the core applications of blockchain technology by enabling the automatic execution of contract rules without human intervention [15], [52], [41], [37]. Once deployed on the blockchain, smart contracts become immutable.

Smart contracts deployed on the blockchain are accessible to everyone and can be viewed by anyone. Despite their deterministic nature, smart contracts currently do not provide a rigorous mechanism to ensure that each instruction is executed solely by authorized roles. This limitation gives rise to access control vulnerabilities, wherein insufficient enforcement of permissions allows unauthorized users to access restricted operations or data [25]. A prominent example is the Parity Multi-Sig Wallet [6], [7], where an attacker leveraged access control vulnerabilities to gain ownership of the contract. By executing the self-destruct operation, the attacker rendered the contract inoperable, resulting in the irreversible freezing of substantial funds.

In recent years, many detection tools have been developed to detect access control vulnerabilities, such as TEETHER [31], Ethainter [14], Spcon [34], Somo [23], and AChecker [26]. Except for Spcon, which focuses on analyzing historical transaction records, the other tools track predefined critical instructions. These tools leverage data flow analysis and taint analysis to trace whether these instructions can be accessed by unauthorized roles.

However, these tools are based on static analysis and heavily rely on predefined rules, which limit their ability to detect vulnerabilities. Specifically, rule-based methods are limited to strictly matching patterns that conform to predefined rules. They lack extensibility and are unable to handle complex scenarios that fall outside the scope of these predefined rules. Unlike other vulnerabilities (e.g., integer overflows), access control vulnerabilities are not solely determined by program structure but are closely tied to the program's actual semantics. For example, in the case of the AC vulnerability `CVE-2019-15079` (see Section II) affecting an Ethereum token, the mapping variable `balanceOf` is associated with various roles (such as `msg.sender`), a situation that previous rule-based approaches fail to cover. However, a typo in the constructor allows an attacker to access this function without authorization. Since static analysis tools are based on predefined rules, they are unable to capture code semantic features, thus limiting their ability to detect access control vulnerabilities in cases like this. The emergence of large language models (LLMs) offers a promising way to fill this gap due to their great potential in code comprehension [48], [30], motivating us to propose a new approach that leverages LLMs for detecting access control vulnerabilities in smart contracts. However, to implement this detection approach, there are several challenges that need to be tackled:

[1]Also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security

[2]Corresponding author

Challenge 1: How to focus LLM attention on context relevant to access control in smart contracts? Smart contracts contain diverse and complex contextual information. However, LLMs often exhibit attention dispersion, focusing on high-frequency patterns due to long-tail data distributions [17], [32] and code complexity. However, context related to access control is sometimes low-frequency and less represented in the training data. As a result, LLMs may overlook this context during the reasoning process across the entire smart contract. **Challenge 2: How to enable LLMs to truly understand the logic underlying access control?** The business logic in smart contracts is human-authored and often contains implicit intentions that are difficult to fully capture. Instead of merely processing individual lines of code, LLMs need to truly understand the underlying logic behind access control.

In this paper, we propose ACTAINT, an LLM-based multi-agent system that detects access control vulnerabilities in smart contracts by two specialized agents: the sink agent and taint agent. The sink agent is responsible for identifying security-sensitive operations–referred to as sinks–that could potentially introduce access control vulnerabilities across the entire smart contract. Meanwhile, the taint agent conducts data flow analysis to determine whether untrusted inputs can propagate to and influence these identified sinks. Our key idea is to leverage two specialized LLM agents that collaboratively perform taint analysis to detect access control vulnerabilities.

To address **Challenge 1**, we employ a static analysis method to guide the Sink Agent toward identifying sinks, thereby focusing the LLM's attention on access control-related operations within the contract. To address **Challenge 2**, we guide the Taint Agent to emulate the behavior of traditional static analysis tools by performing dataflow tracing and taint analysis for each identified sink. In addition, we add a self-evaluation mechanism in the agents to allow LLM to critically assess its own reasoning process and outputs. By leveraging the LLM's native capabilities in code understanding and explanation, this self-evaluation step enhances the reliability of results.

Compared with previous tools, our approach ACTAINT leverages its superior code comprehension abilities to uncover more complex or subtle privilege escalation paths, enabling more accurate detection of access control vulnerabilities. Existing tools are often overly constrained during the detection process, making it difficult to obtain accurate and reliable execution paths of the smart contract. To the best of our knowledge, ACTAINT represents the first approach leveraging LLM-based agents to conduct taint analysis specifically for detecting access control vulnerabilities in smart contracts. The main intuition behind our design is that taint analysis is well-suited for detecting access control vulnerabilities, while LLMs possess strong code understanding capabilities. Our evaluation of ACTAINT is conducted on 3 datasets—the CVE dataset comprising 15 known vulnerable contracts, a collection of 624 real-world smart contracts sourced from the SmartBug Wild dataset [10] and 94 real-world smart contracts collected in [51]. On the first dataset, our approach achieves higher Recall (0.933) and F1-score (0.933) than state-of-the-art tools. On the second dataset, our approach achieves higher Precision (0.622) and F1-score (0.730) than the static analysis tool AChecker.

In summary, we make the following contributions:

- We propose a novel LLM-based multi-agent approach to detect access control vulnerabilities in smart contracts, where each agent is specialized for a distinct reasoning task, including identifying sinks and performing taint analysis. To the best of our knowledge, this is the first work that applies a multi-agent LLM system to perform access control vulnerability detection in smart contracts.
- We conduct experiments on 624 real-world smart contracts and compare the results with the state-of-the-art static analysis tool, AChecker. The results show that ACTAINT achieves higher recall and a higher F1-score.
- Throughout the experimental process, we conducted thorough manual analysis of all smart contracts and their corresponding detection outcomes, thereby offering valuable groundwork for subsequent studies on the fundamental principles of vulnerability mechanisms.

II. RESEARCH MOTIVATION

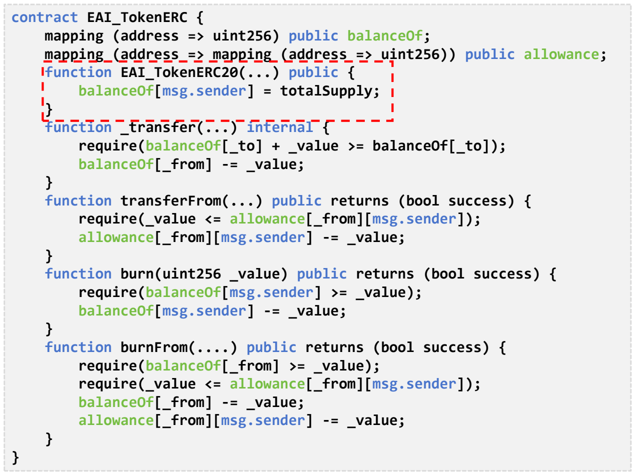
In this section, we use two real-world smart contracts to illustrate the design motivation behind our approach.

As shown in Fig. 1, the first contract coming from `CVE-2019-15079` contains an access control vulnerability. However, the state-of-the-art static tool, AChecker, fails to detect it, which is regarded as a false negative (FN) case.

In this smart contract, the developer implements token management functionality but incorrectly defines the constructor, allowing any user to invoke the function `EAI_TokenERC20` and modify the `balanceOf` variable, thereby introducing an access control vulnerability. AChecker identifies access control state variables by analyzing conditions that check against `msg.sender`, the global variable representing the caller's address. However, the `require` statements in this contract only enforce conditions related to the `mapping` variables `balanceOf` and `allowance`. The tool applies predefined rules for `mapping` type variables, considering only those that either store boolean values exclusively or are used solely within conditional statements. Consequently, AChecker fails to recognize `balanceOf` as a critical access control variable. Handling token management requires an understanding of business logic, which is beyond the capabilities of current static analysis techniques.

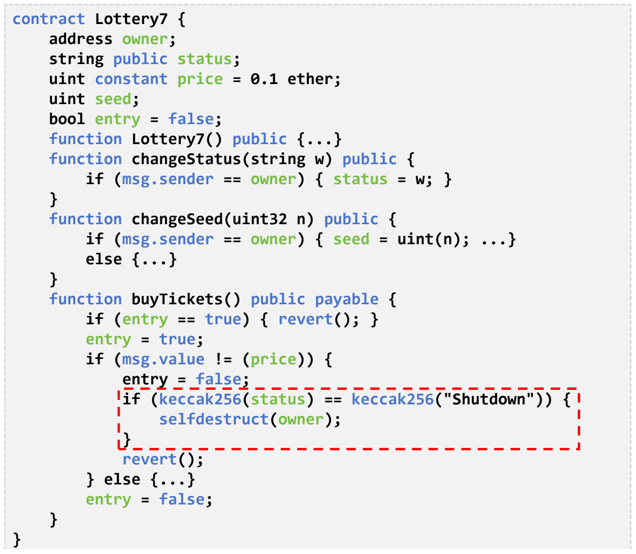
The second smart contract, as shown in Fig. 2, is sourced from the SmartBugs Wild dataset [8]. The function `buyTickets` within this contract includes the instruction `selfdestruct(owner)`. Our manual analysis confirms that this contract is safe and does not contain any access control vulnerabilities; however, AChecker incorrectly reports an access control vulnerability (FP).

This instruction is protected by the condition `keccak256(status) == keccak256("Shutdown")`. Within the contract, only the `owner` can modify the variable `status` via the function `changeStatus()`. However, AChecker incorrectly reports



```
contract EAI_TokenERC {
    mapping (address => uint256) public balanceOf;
    mapping (address => mapping (address => uint256)) public allowance;
    function EAI_TokenERC20(...) public {
        balanceOf[msg.sender] = totalSupply;
    }
    function _transfer(...) internal {
        require(balanceOf[_to] + _value >= balanceOf[_to]);
        balanceOf[_from] -= _value;
    }
    function transferFrom(...) public returns (bool success) {
        require(_value <= allowance[_from][msg.sender]);
        allowance[_from][msg.sender] -= _value;
    }
    function burn(uint256 _value) public returns (bool success) {
        require(balanceOf[msg.sender] >= _value);
        balanceOf[msg.sender] -= _value;
    }
    function burnFrom(....) public returns (bool success) {
        require(balanceOf[_from] >= _value);
        require(_value <= allowance[_from][msg.sender]);
        balanceOf[_from] -= _value;
        allowance[_from][msg.sender] -= _value;
    }
}
```

Fig. 1: An Example of False Negative in the Contract `EAI_TokenERC`



```
contract Lottery7 {
    address owner;
    string public status;
    uint constant price = 0.1 ether;
    uint seed;
    bool entry = false;
    function Lottery7() public {...}
    function changeStatus(string w) public {
        if (msg.sender == owner) { status = w; }
    }
    function changeSeed(uint32 n) public {
        if (msg.sender == owner) { seed = uint(n); ...}
        else {...}
    }
    function buyTickets() public payable {
        if (entry == true) { revert(); }
        entry = true;
        if (msg.value != (price)) {
            entry = false;
            if (keccak256(status) == keccak256("Shutdown")) {
                selfdestruct(owner);
            }
            revert();
        } else {...}
        entry = false;
    }
}
```

Fig. 2: An Example of False Positive in the Contract `Lottery7`

this as a vulnerability due to its inability to recognize the conditional dependency on variable `status`. Furthermore, it fails to capture the relationship between the variables `status` and `owner`. Since `status` is of type `string`, it falls outside the scope of AChecker's analysis. This limitation highlights the constraints of static analysis tools, whose detection rules are predefined by the author and lack full coverage of all security conditions.

These two examples illustrate that while static analysis tools offer advantages in vulnerability detection, they lack an understanding of business logic and program context. This limitation motivates us to propose a detection approach that integrates static analysis with LLMs.

III. Our approach: ACTaint

A. *Overview of* ACTaint

In this section, we present the workflow of our approach, as shown in Figure 3. We begin by analyzing a given smart contract to extract critical variables associated with access control operations and construct a Critical Variable Dependency Graph (CVDG). Additionally, we detect critical instructions that are potentially involved in access control logic. Next, we utilize an LLM-based agent to detect potential sinks, guided by the constructed CVDG and critical operations. Finally, we utilize the LLM to perform a simulated taint analysis by tracing the data flow from sources to sinks, enabling us to detect the presence of access control vulnerabilities in the smart contract. Taint analysis is typically formalized as a triplet $<source, sink, sanitizer>$. It traces whether data can flow from a source to a sink without passing through any sanitizers. In this paper, a source refers to untrusted external input, a sink is a security-sensitive operation that may cause access control vulnerabilities, and a sanitizer represents a conditional check that constrains the data flow. The reason we adopt a multi-agent approach is that it can effectively decompose complex tasks, thereby overcoming the limitations of single prompts in handling long contexts and intricate logic. Specifically, we divide the task into a sink agent and a taint agent, whose results complement each other. This design functions as a form of memory, mitigating the issue of attention distraction in large models caused by excessive contextual information.

B. *Step 1: Build Critical Variable Dependency Graph*

1) Preparation: The input is a fully preprocessed and compilable smart contract. We select an appropriate Solidity compiler version based on the version specified in the contract using `solc-select` [11]. Then, we start our analysis.

2) Identify Critical Variables: We utilize Slither [9] to extract state variables from a smart contract and analyze condition check expressions to determine which variables are referenced in enforcing access control mechanisms. Condition check expressions include `if`, `assert`, and `require` statements, which influence the flow of execution in a smart contract. We convert the smart contract's source code into intermediate representations (IRs) and analyze them to systematically identify critical variables. Critical variables are state variables that are either used in conditional checks or have the type `address`. Additionally, binary expressions that perform comparisons and return boolean values are classified as condition check expressions. While some binary expressions involve arithmetic operations (e.g., addition, subtraction, multiplication, and division), they are not related to condition checks and thus are excluded from our analysis. State variables of type `address` are directly classified as critical, as they consistently store the addresses associated with specific roles. Through this process, we obtain the set of critical variables, denoted as S_{cv}.

3) Build Critical Variable Dependency Graph: After identifying the critical variables, we construct the Critical Variable

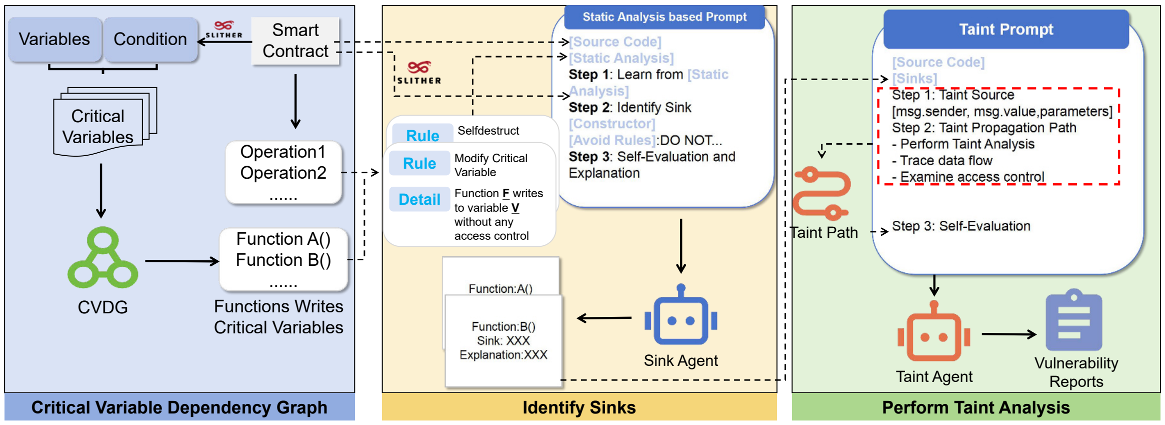
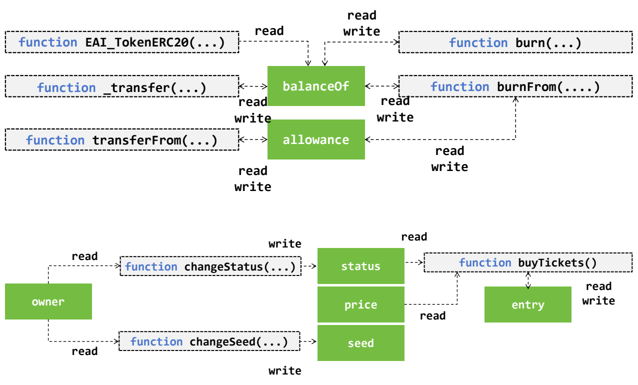


Fig. 3: Th

Dependency Graph (CVDG), a directed graph that rep
the dependencies between critical variables and the fu
or other variables they influence. We systematically a
each function in the contract to determine whether it m
any of these variables. This analysis is essential, as cl
to critical variables can directly impact access control
anisms. At this stage, for a given critical variable
a function F modifies its value, we denote that func
writes the critical variable CV. Consequently, we es
a directed edge from the function F to the critical v
CV. Similarly, we say that a critical variable CV is *re*
function F if CV is involved in a conditional check expres
within F; in this case, we add a directed edge from the criti
variable CV back to the function F. By the way, we do n
consider the constructor when constructing the graph,
it is executed only once during deployment. We convert t
function F into intermediate representations (IRs) and perfo
data flow analysis to determine whether it modifies any criti
variable CV. Additionally, we examine whether the functi
contains condition check statements and identify the associated state variables. Given that Solidity-specific modifiers also play a crucial role in access control, we analyze whether they reference state variables as well.

Finally, we construct a Critical Variable Dependency Graph (CVDG). To provide a concrete illustration of our approach, we revisit the motivating examples in Section II. In the contract EAI_TokenERC, $S_{cv} = \{balanceOf, allowance\}$. The functions EAI_TokenERC20, _transfer, burn, and burnFrom *writes* to critical variable balanceOf, while function transferFrom and burnFrom *writes* to critical variable allowance. Variable balanceOf is *read* in functions _transfer, burn and burnFrom. Variable allowance is *read* in function transferFrom. Fig. 4a illustrates the CVDG for contract EAI_TokenERC. In this figure, dashed boxes represent functions, green rectangles denote critical variables, and directed edges indicate dependency relationships between them. Similarly, in the contract Lottery7, $S_{cv} = \{owner, status, price, seed, entry\}$. By following the same procedure, we can obtain the corresponding CVDG, illustrated in Fig. 4b.



(b) CVDG of Contract Lottery7

Fig. 4: CVDG of Motivation Examples

C. Step 2: Sink Agent

In this step, we build the Sink Agent, which plays a central role in identifying potential sinks. The agent integrates critical operations, static analysis outputs from the previous step and LLMs to automatically detect and evaluate these sinks.

1) Critical Operation: Critical operations refer to certain operations that, if not properly protected by access control mechanisms, can be accessed by unauthorized users, thereby leading to access control vulnerabilities. In our approach, We define critical operations as potential sinks. We adopt the three critical operations from a prior study[26], [23], which are as follows:

- **Critical Variable Modification.** As discussed in Section III-B, critical variables are closely tied to the access control logic in smart contracts. Improper modification of these variables can directly compromise access control enforcement and lead to privilege escalation or bypass.
- **Destroy Contract.** The selfdestruct(address recipient) is a basic EVM instruction that can destroy the whole smart contract. As a result, this is a critical operation and is typically restricted to the contract owner.
- **Low-level Call.** In solidity, address type variable has three member functions to call functions externally. In smart contracts, low-level calls can be implemented by

using EVM instructions such as `call`, `staticcall`, and `delegatecall`. The `delegatecall` differs from `call` in that it executes using the data and context of the caller (`msg.sender`), rather than that of the target contract. It is highly dangerous, as it effectively executes the callee contract's code using the caller's data and context. The `callcode()` was deprecated in Solidity version 0.5.0 and replaced by `delegatecall`. Therefore, it is not discussed in this paper.

Additionally, we also consider the other three critical operations, as our observations indicate that they can also lead to access control vulnerabilities:

- **Tx.origin.** The global variable `tx.origin` refers to the original sender of the transaction that initiated the call to the contract, differing from `msg.sender`, which refers to the current caller. An attacker can spoof `tx.origin` to bypass the check condition instructions.
- **Transfer**. In a smart contract, basic transfer instruction includes `transfer`, `send`, and `call`. As one of the fundamental functionalities of smart contracts, transferring Ether from a contract is considered a sensitive operation.
- **Assembly.** The assembly block is a construct in smart contracts that allows direct manipulation of memory. By bypassing high-level Solidity abstractions, it provides greater flexibility and execution efficiency. However, due to its ability to access and modify low-level data structures, it requires proper access control.

2) Scope of Analysis: Relying on code slicing alone is insufficient for detecting access control vulnerabilities in smart contracts. We revisit the motivating examples to illustrate this limitation. In the contract `EAI_TokenERC`, shown in Fig.1, analyzing whether the function `EAI_TokenERC20` modifies the variable `balanceOf` securely requires a comprehensive understanding of `balanceOf` across the entire contract, including its usage in other functions. The variable `balanceOf` is also modified by the functions `_transfer`, `burn`, and `burnFrom`. Additionally, since function `burnFrom` writes the critical variable `allowance`, it is also necessary to consider `allowance` in our analysis. Similarly, in the contract `lottery7`, as shown in Fig. 2, assessing the security implications of `selfdestruct` requires analyzing variables such as `entry`, `price`, and `status`, where `status` is guarded by `owner`-based access control.

Therefore, based on our findings, analyzing the entire smart contract, rather than relying solely on program slices, enables more accurate detection of access control vulnerabilities. However, when analyzing an entire smart contract, the attention of large language models (LLMs) may become dispersed, making it difficult to focus on access control–specific information. In particular, access control terms tend to be relatively low-frequency within smart contracts, which further limits the model's ability to recognize them without explicit guidance. Based on our previous observations, we adopt a static analysis based prompting strategy.

3) Static Analysis: To identify modifications to critical variables, we traverse the CVDG obtained in Section III-B to determine whether there exists a function `F` that writes to a critical variable `CV`. We refer to such functions as those that write to critical variables without any access control.

We revisit the motivating examples in Section II. As shown in Fig. 4a, we can find that the function `EAI_TokenERC20` is the only one without any incoming edges in the CVDG. Therefore, we can conclude that the function `EAI_TokenERC20` is the sole function in contract `EAI_TokenERC` that writes critical variable `balanceOf` without any access control. Similarly, as shown in Fig. 4b, each function writes to critical variables under access control.

For the remaining critical operations, we identify and record them by statically extracting their corresponding low-level instructions. Additionally, prior studies [26] have identified specific EVM instructions that, if not properly handled, may introduce access control vulnerabilities. However, these approaches heavily rely on manually predefined rules, which may limit their generalizability and adaptability to diverse contract patterns.

4) Static Analysis based Prompt: Based on these static analysis results, we propose a prompt strategy, as shown in Fig. 3. To enhance the reasoning capability of LLMs, we adopt the Chain-of-Thought (CoT) prompting strategy [48]. First, we collect static analysis results and formulate a set of rules to constrain the LLM's attention to access-control-relevant code regions. For example, Function `F` writes to variable `V`. These rules are expressed in natural language to better align with the LLM's strengths in language understanding and to effectively guide its focus toward critical semantics.

Then, based on the insights from the previous analysis, we guide the LLM to identify sink operations relevant to access control. Based on our observations, we identify several scenarios that may lead the LLM to generate hallucinations. One common source of hallucination arises when the LLM misidentifies the constructor function. It may incorrectly treat a regular function as the constructor or overlook the actual constructor. This issue often stems from differences in constructor declaration syntax across Solidity versions, as well as factors like function naming and case sensitivity. To address this issue, we statically identify the true constructor function using Slither and explicitly guide the LLM to treat only this function as the constructor. LLM sometimes tends to perform speculative reasoning by default, often introducing hypothetical "`if`..." conditions that may lead to incorrect conclusions. Therefore, we instruct the LLM to avoid using phrases such as `if`, `might`, `could`, or any uncertain language.

5) Self-Evaluate: Existing research [20], [22], [21] has found that LLMs can leverage their inherent capabilities to perform reasoning and self-assessment. To further enhance the reliability of taint analysis, we incorporate a self-explanation step, prompting the LLM to explicitly articulate its reasoning about the taint propagation path. This step leverages the LLM's internal knowledge to critically assess the plausibility of each path, allowing it to filter out spurious or semantically invalid taint flows. As a result, we ask the LLM to self-evaluate its reasoning process and provide a formal explanation. This step helps the LLM to review and reflect on its own reasoning, enabling it to identify potential errors or inconsistencies. Moreover, a clear explanation can provide the Taint Agent with a better understanding of the underlying logic, thereby improving its ability.

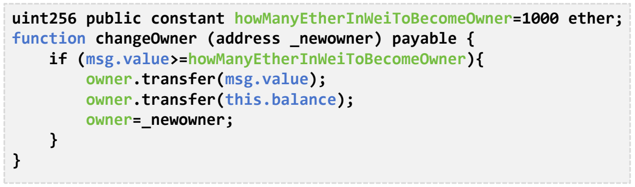
D. Step 3: Taint Agent

In this step, we employ the Taint Agent to perform taint analysis and determine whether untrusted inputs can reach the sinks identified in Step 2. As is widely recognized, taint analysis is well-suited for detecting access control vulnerabilities. However, traditional static analysis faces significant limitations when applied to real-world execution paths, primarily because it does not simulate the actual execution of smart contracts and often relies on logic predefined. For example, although AChecker claims not to rely on predefined rules, its handling of complex variables such as `mapping` is still based on the predefined algorithm from the authors' observations.

Unlike traditional static analysis, LLMs demonstrate a stronger capability to understand the semantic logic of smart contracts. Therefore, we assign the task of taint analysis to the LLM agent, enabling it to handle more complex scenarios without being constrained by rigid, predefined rules. This allows the LLM to more flexibly explore potential data flow paths from sources to sinks, leading to more effective detection of access control vulnerabilities. The prompt used for taint analysis is illustrated in Fig. 3. In this agent, we adopt the CoT [48] to enable the LLM to simulate traditional taint analysis step by step.

1) Taint Source: Taint analysis begins with identifying taint sources, a critical first step in the taint analysis for smart contracts. We guide the LLM to detect untrusted inputs–such as `msg.sender` and function parameters–that may influence security-critical operations. A key distinction arises between `tx.origin` and `msg.sender`, as they embody different security semantics. While `msg.sender` represents the immediate function caller, `tx.origin` traces the transaction's originating external account, which has historically introduced vulnerabilities. As noted earlier, `tx.origin` should only be used in specific cases (e.g., $tx.origin == msg.sender$). Unlike rigid rule-based methods, our approach leverages the LLM's dynamic reasoning to identify context-dependent taint sources (e.g., `msg.value`, `tx.origin`, or function parameters) for each sink. This autonomy enhances scalability and adaptability in real-world contract analysis.

2) Taint Propagation Path: After identifying a taint source, we guide the LLM to analyze the taint propagation path, tracing how the untrusted data flows through the program to reach the sink without a sanitizer. In the context of smart contracts, sanitizers refer to a program constructor that effectively constrains the tainted data before it can reach the sink. These typically include `require` statements, conditional branches (e.g., `if` statements that guard critical logic), and function modifiers. Such elements can enforce preconditions or introduce control-flow barriers that prevent unsafe execution paths. In this step, we explicitly emphasize that only actual execution paths should be considered, as LLMs tend to perform hypothetical reasoning that may not align with real contract behavior. While LLMs are not capable of executing smart contracts, their semantic understanding allows them to reason about implicit or non-trivial data flows and infer feasible execution paths, which are often missed by traditional static analysis techniques.



```
uint256 public constant howManyEtherInWeiToBecomeOwner=1000 ether;
function changeOwner (address _newowner) payable {
    if (msg.value>=howManyEtherInWeiToBecomeOwner){
        owner.transfer(msg.value);
        owner.transfer(this.balance);
        owner=_newowner;
    }
}
```

Fig. 5: An Example of True Negative

3) Self-Evaluate: Similar to the Sink Agent, we apply a self-evaluation mechanism to assess whether the inferred execution paths are valid. Leveraging the LLM's domain knowledge, this step helps filter out obviously incorrect paths and improves the overall reliability of the taint analysis.

IV. Experimental Setup

A. Research Questions

We evaluate ACTaint by answering the following research questions (**RQs**):

RQ1: *How does* ACTaint *perform in detecting access control vulnerabilities compared to existing approaches?*

RQ2: *What is the contribution of key components in our approach to the overall performance improvement?*

RQ3: *How well does* ACTaint *work when applied to different LLMs?*

B. Dataset

Our evaluation dataset is composed of three sources: CVE, AChecker [26] and ACFix [51].

1) CVE: The first dataset used in our experiments is the CVE dataset, containing 15 smart contracts with known access control vulnerabilities. Each contract is linked to a specific Common Vulnerabilities and Exposures (CVE) identifier, ensuring that the dataset reflects real-world security issues and serves as a valuable resource for testing vulnerability detection performance. In this dataset, we evaluate vulnerabilities at the function level. A detection is considered a TP only if both the location and the root cause are correctly identified. Otherwise, it is classified as a FP. All these contracts contain vulnerabilities and have existing CVE reports. If a vulnerability documented in the CVE report is not detected by our tool, it is counted as a FN.

2) Smartbugs Wild: The second dataset is derived from AChecker's vulnerability detection results on the SmartBugs-Wild dataset [10], which comprises 47,398 smart contracts collected from the Ethereum blockchain. While AChecker reported 624 contracts, only 60 of these reports were manually verified in their original work. To enable a more comprehensive and reliable comparison in our study, we manually examined all 624 reported contracts and selected them as our second dataset. We manually analyzed each smart contract. Here, to facilitate the statistical analysis of results, we localized vulnerabilities at the function level. In total, we identified 481 access control vulnerabilities through manual analysis. Detection reports that correspond with our analysis are counted as TP, whereas mismatches are regarded as FN. Functions reported beyond those analyzed are classified as FP. To annotate the ground-truth labels for all 624 contracts from SmartBugs, two researchers with over one year of smart contract experience independently annotated them, each taking about a week. Disagreements occurred in approximately 10% of cases, which were resolved by a third expert with over seven years of experience who reviewed the contracts and made final judgments. We further identified 41 cases that correspond to what AChecker referred to as intended behavior—that is, cases that resemble access control vulnerabilities in principle, but are actually permitted by the contract's business logic. These cases are considered secure in our experiment. We select one example for illustration [13]. In Fig. 5, anyone can invoke the function `changeOwner` and spend more than 1000 ETH to change the value of `owner`. Note that our evaluation focuses on root causes; therefore, multiple reports corresponding to the same root cause are counted as a single instance. If the modifier `onlyOwner` contains an access control vulnerability, any function protected by `onlyOwner` is not counted regardless of whether it is reported; a TP is recorded only when the modifier `onlyOwner` itself is precisely identified and its root cause correctly explained. We do not count the same function multiple times. Different functions that correspond to the same execution path, for example, functions invoked via the `fallback`, are treated as a single case. Due to the high computational cost of applying LLM-based methods at scale, we don't run our approach on all 47,398 smart contracts.

3) ACFix Dataset: The third dataset, built by the authors of ACFix, is by far the first large benchmark dataset specifically for access control vulnerabilities[51]. The dataset comprised of 113 smart contracts, including 19 contracts that overlapped with our previous database and one is considered by us to have no actual access control vulnerability. After filtering out these overlapping cases, we obtained a set of 93 unique smart contracts. Among these 93 contracts, a total of 99 access control vulnerabilities were identified. The contract within this dataset was largely derived from a combination of audit reports and GitHub blogs. Successfully compiling these contracts is a significant challenge, as many required specific version dependencies. Furthermore, several vulnerabilities had already been patched. We only obtained the post-fix versions of the contracts and were unable to access the original vulnerable versions. After investigation, we retained 46 contracts that compiled successfully. For the remaining 47 contracts that failed to compile, we conducted experiments by removing Step 1. As a result, we applied a no-static module to run these contracts that could not be compiled. We record TP and FP cases using the methodology applied to the second dataset.

TABLE I: The Performance on the CVE Dataset

	TP	FP	FN	Precision	Recall	F1
ACTAINT	14	1	1	0.933	0.933	0.933
AChecker	12	0	3	1.000	0.800	0.889
GPTLens	6	2	9	0.750	0.400	0.552

C. Baselines

To assess the effectiveness of our approach, we compare ACTAINT against two representative baselines. The first is AChecker, a state-of-the-art static analysis-based vulnerability detection tool specifically designed for smart contracts. According to the AChecker paper, it has been evaluated against other static analysis tools such as Slither, Mythril, and SPCon, and has demonstrated superior performance in detecting access control vulnerabilities in smart contracts. To the best of our knowledge, there are currently no LLM-based tools specifically designed for detecting access control vulnerabilities in smart contracts. Therefore, we include GPTLens [28] as a baseline, which is a general-purpose vulnerability detection framework utilizing large language models (GPT-4 in our study). For GPTLens, which is not specifically designed for access-control vulnerabilities, we evaluate only those reported issues that are related to access control and ignore others. Moreover, GPTLens performs self-criticism; if during this process it self-rejects a finding and typically assigns a score below 4 (based on our observation), the finding is not considered.

D. Experimental Setup

The implementation of ACTAINT is based on a combination of Slither, a static analysis framework for Solidity, and a large language model (LLM) accessed via the GPT API. All LLM-based analyses in our experiments, including those conducted using GPTLens, were performed through API calls. To balance cost and effectiveness when using proprietary models, we select GPT-4o as the LLM in our approach due to its relatively low cost and strong performance. For our LLM API configuration, we set the $temperature$ to 0 and top_p to 0.9, which is a standard parameter setting as in other studies [44]. This configuration ensures highly deterministic outputs for precise vulnerability identification while maintaining diversity in generated samples.

All experiments were conducted on a server equipped with an Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz, with 64 logical processors. For AChecker, we obtained the detection results and relevant datasets directly from the authors via email correspondence. Therefore, in this paper, we directly reference their experimental results.

V. EXPERIMENTAL RESULTS

A. RQ1: The Effectiveness of ACTAINT

To answer RQ1, we perform experiments on the first and second datasets. Tables I present the results and comparative performance of the approaches on the CVE dataset, respectively. Table II shows the performance of the approaches on 624 smart contracts from the SmartBug dataset.

TABLE II: The Performance on SmartBug Dataset

	TP	FP	FN	Precision	Recall	F1
ACTAINT	425	258	56	0.622	0.884	0.730
AChecker	369	315	112	0.539	0.767	0.633
GPTLens	155	242	326	0.390	0.322	0.353

TABLE III: The Performance on ACFix Dataset

	TP	FP	FN	Precision	Recall	F1
ACTAINT	36	17	63	0.679	0.364	0.474
AChecker	2	0	97	1	0.020	0.040
GPTLens	22	16	77	0.579	0.222	0.321

In this first dataset, we compare ACTAINT's performance with AChecker and GPTLens. Here, we localize vulnerabilities at the function level. Unlike AChecker, we count multiple reports with the same root cause as a single report, emphasizing our focus on the underlying root cause. As shown in Table I, we observe that ACTAINT achieves 14 TPs, which is 2 more than AChecker. We further analyze the two additional TPs identified by ACTAINT, which correspond to `CVE-2019-15079` and `CVE-2020-17753`. The contract `CVE-2019-15079` is the motivating example discussed in Section II. It involves a complex variable `balanceOf`, which cannot be handled by static analysis approaches. The vulnerability in contract `CVE-2020-17753` stems from the misuse of `tx.origin`, which AChecker fails to identify. In the remaining contracts, the majority of vulnerabilities are associated with modifications of critical variables, and are effectively identified by both ACTAINT and AChecker. A comparative analysis with GPTLens indicated that the latter produced a greater number of FPs and FNs, alongside a reduced count of TPs. This discrepancy is largely attributable to GPTLens's limited efficacy in addressing access control vulnerabilities. Furthermore, our findings suggest a notable deficiency in GPTLens's ability to identify vulnerabilities associated with the modification of critical variables.

As shown in Table II, ACTAINT achieves higher precision, recall and F1-score than AChecker and GPTLens because it can detect more TP. ACTAINT detects some vulnerabilities that were not reported by AChecker. Among approximately 74 files, ACTAINT detects more true positives than AChecker. These TPs can be roughly categorized into business logic issues, the initialization of complex variables beyond the address type and the incorrect use of `tx.origin`.

In the third dataset, some cases only provided code snippets. We try to find the source code on Etherscan [4] and downloaded the relevant dependencies to ensure correct compilation. We classify the results as TP or FP, following the same criteria as in the second dataset. As show in Table III, ACTAINT achieves more Precision, Recall and F1-score than GPTLens.

AChecker requires bytecode to operate. In this dataset, due to compilation limitations, it was not possible to run all experiments. Therefore, we directly used the results from ACFix and performed manual secondary verification locally. Due to this limitation, it is not comparable on this dataset.

Summary: ACTAINT demonstrates higher precision in detecting access control vulnerabilities in smart contracts compared to baselines, achieving superior Precision, Recall and F1-score.

B. *RQ2: Effectiveness of key components in* ACTAINT.

To answer RQ2, we conduct an ablation study to assess the individual contributions of the static analysis module and the taint agent to the overall detection performance of ACTAINT. We build two variants of ACTAINT by removing two key modules, which are as follows:

- **ACTAINT-NoStatic**: We remove the static analysis module from our tool, denoted as ACTAINT-NoStatic. Specifically, as illustrated in Fig. 3, in Step1 (Critical Variable Dependency Graph), we utilize the static analysis framework Slither to extract critical variables and subsequently construct CVDG based on these critical variables. In addition, we also identify critical operations that may lead to access control vulnerabilities in this step. This static analysis information is removed before feeding into Step2 (Identify Sinks) to construct the prompt. ACTAINT is then run without any static information, starting directly from the Sink Agent in Step 2.
- **ACTAINT-NoTaint**: For the non-taint model, we aim to assess the contributions brought by the taint analysis process, we therefore retained the first two steps (Step1, Step2) and removed Step3 (Taint Analysis). The output of Step2, Identify Sinks, is treated as the final result.

We use the second dataset for the ablation study, and the results are shown in Table IV. For no static, precision drops to 0.538, recall to 0.281, and F1-score to 0.369. The decline can be attributed to the absence of static analysis, which substantially reduces the ability of ACTAINT to identify sinks. As a result, TP decreases and FN increases. Although FP is also reduced in this process, the overall effectiveness still deteriorates. For no taint, precision drops to 0.281, recall rises to 0.940, and F1-score decreases to 0.432. This is because, without the taint module, we directly treat the sinks identified by ACTAINT as vulnerabilities. As a result, TP increases and FN decreases, but FP rises substantially. Consequently, recall improves, while the other two metrics decline.

Summary: The components of ACTAINT, including static analysis and the taint agent, both contribute positively to the overall performance. Moreover, compared to the taint agent, static analysis makes a greater contribution.

C. *RQ3: Effectiveness of* ACTAINT *in different LLMs.*

To answer RQ3, we examine the adaptability and robustness of ACTAINT when deployed with different LLM backbones. Specifically, we aim to understand whether ACTAINT's effectiveness depends heavily on the choice of LLM or whether

TABLE IV: The Performance Comparisons in the ablation study

Variant	Precision	Recall	F1
ACTAINT	0.622	0.884	0.730
ACTAINT-NoStatic	0.538	0.281	0.369
ACTAINT-NoTaint	0.281	0.940	0.432

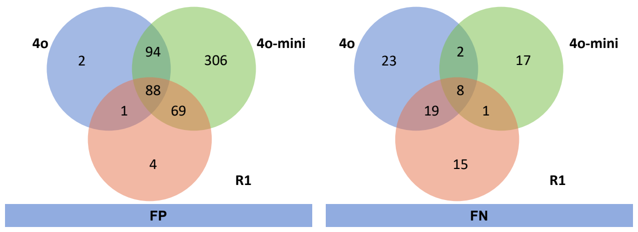


Fig. 6: FP and FN Venn for Different LLms

its overall methodology generalizes well across model architectures. For this purpose, we integrate the LLM-based components of our approach into two representative models: GPT-4o-mini, a commercially available model developed by OpenAI, and DeepSeek-R1, an open-source model that undergoes reinforcement learning-based post-training. These two models allow us to evaluate the adaptability of ACTAINT across different LLM backbones with varying architectural characteristics and training methodologies. To ensure a controlled and fair comparison, we retain the same prompt design, pipeline structure, and evaluation datasets across all settings. Only the LLM backbone is substituted.

We use the second dataset. The results, shown in Table V, reveal significant performance differences among the three LLMs: GPT-4o-mini, DeepSeek-R1, and GPT-4o.

For 4o-mini, it generates a large amount of output, most of which are FPs, making the statistics a hard task. However, the reasoning and reporting of 4o-mini do not seem to accurately distinguish whether it truly identifies affected functions. For instance, in R1, it can correctly identify the taint path of an affected function. Beyond this, it may also hallucinate and produce non-existent function names. We conducted a careful analysis and, instead of calculating affected functions, treat its incorrect judgments as FPs. Overall, due to its limited capability, 4o-mini is not suitable for use. In this section, we only discuss its ability.

We construct Venn diagrams for files (rather than cases) that generate FP and FN, as shown in Fig. 6. We then analyze the results and find that: (1) Due to differences in model capabilities, the final results produced by ACTAINT vary across models. GPT-4o-mini struggles to understand the underlying logical structure of smart contracts and does not strictly follow the logic prescribed in the prompt, which results in a large number of FPs. (2) GPT-4o achieves performance similar to DeepSeek R1. Since R1 is a reasoning-based model, it performs better in comparison. (3) Regarding FN, they are primarily caused by limitations in the LLMs' capabilities.

TABLE V: The Performance of Different LLMs

LLM	Precision	Recall	F1
GPT-4o-mini	0.145	0.938	0.251
Deepseek-R1	0.652	0.902	0.757
GPT-4o	0.622	0.884	0.730

Summary: The core ideas of ACTAINT are applicable to other large language models, and different LLMs have distinct impacts on the ACTAINT overall performance.

VI. DISCUSSION

In this section, we analyze the false positive and false negative cases identified by ACTAINT, evaluate its efficiency, and discuss potential threats to the validity of our study. The causes of FP and FN cases are rather diverse. We do not discuss each category individually. Instead, we select two representative category for illustration and more detailed information can be found in [12].

A. False Positive Analysis

To better understand FPs, we conduct a detailed analysis of each FP case. We present two representative categories.

First, we observe that ACTAINT can still **misinterpret certain business logic**. We identify 60 such files and provide a detailed analysis.Normal operations include Ether transfers and token management, such as transferring Ether to an account in exchange for tokens or performing withdrawals. These steps include legitimate token creation operations that involve complex transfer sequences and modifications of critical variables. In these situations, ACTAINT sometimes erroneously identifies these legitimate operations as vulnerabilities. We preliminarily attribute this to the limitations of the LLM, and in future work, we aim to further improve its understanding capabilities.

Second, we also observe that LLMs may still struggle to **accurately interpret implicit access control enforced by complex logic in smart contracts**. This includes failing to understand `msg.value`, misinterpreting `ecrecover` signature verification and other cases. We find 22 files and select one for a detailed explanation [3]. Although LLMs significantly outperform traditional static approaches in terms of semantic understanding, there remains a gap between their current capabilities and full comprehension of diverse contract behaviors. For instance, as shown in Fig. 7, the function `takeAGuess` uses critical operation `selfdestruct` to destroy the smart contract. However, the corresponding contract `LuckyNumber`, implements a number-guessing game: users submit a guess along with a small amount of ETH and if the guess matches a pre-defined value, the contract transfers a reward and then terminates. While the code structure is relatively simple, the business semantics behind this logic pose challenges for LLMs. In this case, ACTAINT fails to capture the intended purpose of the function and instead outputs a shallow path involving `_myGuess`, `winningNumber` and `selfdestruct`, without recognizing their semantic relationship within the game logic.

```
contract LuckyNumber {
    address owner;
    uint winningNumber = uint(keccak256(now, owner)) % 10;
    function takeAGuess(uint _myGuess) public payable {
        require(msg.value == 0.0001 ether);
        if (_myGuess == winningNumber) {
            msg.sender.transfer((this.balance*9)/10);
            selfdestruct(owner);
        }
    }
}
```

Fig. 7: FP: Implicit access control enforced by complex logic

```
contract EqvcTokens is StandardToken, Ownable {
    function EqvcToken(address admin) public {
        totalSupply = INITIAL_SUPPLY;
        // Mint tokens
        balances[msg.sender] = totalSupply;
        // Approve allowance for admin account
        adminAddr = admin;
    }
}
```

Fig. 8: FN 1: Constructor function

B. False Negative Analysis

To gain deeper insights into the causes of FNs, we conduct a detailed analysis of each FN case. We also identify two representative categories for FN. First, our analysis reveals that LLMs struggle to **properly reason about constructor and initialization functions** (see Section III). While we address this limitation through static analysis that explicitly identifies constructor boundaries, the LLMs persistently misinterpret these functions as potential sinks during taint analysis. This occurs despite clear prompt cues that constructors should be treated as trusted initialization points rather than vulnerability sources.

We identify 7 files and select one for a detailed explanation [1]. As shown in Fig. 8. We can observe that the contract `EqvcTokens` includes a function named `EqvcToken` intended to initialize state variables. However, due to a typographical error by the developer—omitting the final 's'—the function name does not exactly match the contract name and therefore it is not treated as a constructor by the Solidity compiler. Despite explicitly providing constructor-related static analysis results in the prompt, the LLM fails to identify this mismatch and mistakenly treats the function as a constructor, ultimately leading to an FN.

Second, we also find that the LLM **may miss some taint paths during the analysis process.**. We find 5 files and select one for a detailed explanation [2]. As shown in Fig. 9, in the contract `OneYearDreamTokensVestingAdvisors`, the function `withdrawTokens` is declared as `private` and guarded by the `whenInitialized` modifier. This ensures that withdrawals cannot be invoked directly by external users and can only occur after the contract has been initialized. However, external users can still trigger `withdrawTokens` via the `fallback` function. This taint path indeed exists and allows an unauthorized user to potentially destroy the contract. Unfortunately, this path is missed, resulting in an FN.

```
contract OneYearDreamTokensVestingAdvisors {
    function () external {
        withdrawTokens();
    }
    function withdrawTokens() private
                whenInitialized{
        ......
        if (dreamToken.balanceOf(this) == 0) {
            selfdestruct(withdrawalAddress);
        }
    }
}
```

Fig. 9: FN 2: Missing taint paths

C. Analysis on Token Cost and Execution Time

To quantify the cost and runtime of a full detection process, we measured both token consumption and execution time on our dataset. For the first dataset, ACTaint processed 950,158 tokens across all contracts in a total of 340.827 seconds, corresponding to an average of 22.722 seconds and 6,010.533 tokens per file. The processing time ranged from 0.424 to 72.245 seconds per file, and the number of tokens per file ranged from 0 to 12,994. For the second dataset, ACTaint processed 4,389,860 tokens across all contracts in a total of 12,075.214 seconds, corresponding to an average of 19.126 seconds and 7,016.546 tokens per file. The processing time ranged from 0.371 to 111.556 seconds per file, and the number of tokens per file ranged from 0 to 39,769. In our method, the number of LLM invocations is determined by both the number of contracts within a file and the number of identified sinks. Additionally, we explicitly generate intermediate reasoning steps, which contribute to the increased runtime. Prior studies have shown that incorporating such reasoning steps can significantly improve the effectiveness of smart contract vulnerability detection [16].

D. Threats to Validity

1) Threats to external validity: Our evaluation is conducted on a benchmark comprising 624 smart contracts. While the dataset encompasses a range of practical cases, it may not comprehensively represent the full landscape of access control vulnerabilities. Although we utilize multiple large language models such as GPT-4o, GPT-4o-mini, and DeepSeek-R1, these models do not cover the entire variety of existing LLM architectures. Moreover, our static analysis techniques and prompt designs may have limited generalizability to all access control scenarios in smart contracts. In our dataset, some contracts use outdated versions or legacy syntax that Slither cannot handle, resulting in **compilation failures**. We manually adjusted these contracts to address this issue.

2) Threats to internal validity: We select our dataset based on AChecker—the state-of-the-art tool for detecting AC vulnerabilities—which identifies 624 vulnerable contracts in the SmartBugs Wild dataset. Only 60 of these were manually verified by AChecker's study. We include all 624 contracts in our evaluation to ensure broad coverage of diverse contract types. However, AChecker may not encompass all contract types. This is a threat in that Smartbugs Wild may still include other smart contracts with access control vulnerabilities that have yet to be discovered. Furthermore, the effectiveness of other studies on Smartbug Wild for detecting ac vulnerabilities is limited [19]. The inherent tendency of LLMs to hallucinate can introduce a degree of variability into ACTAINT's results. In our approach, we mitigate hallucinations by providing construct-level information and encouraging the LLM to perform self-explanation. However, it remains fundamentally difficult to eliminate the associated risks and inherent limitations of the model. In this paper, the identification of access control vulnerabilities relies on manual analysis, which introduces subjectivity and may affect the consistency and objectivity of the vulnerability assessment.

3) Potential vulnerability: We observe certain potential vulnerabilities that, despite not leading to tangible loss, reveal flaws in access control. In the second dataset, one case involves a modifier employing `tx.origin`, which is unreachable, and another involves an unrestricted `owner.transfer`. Although the transfer is directed to the contract owner, this represents an improper access control pattern. ACTAINT is capable of detecting these potential vulnerabilities; however, they are not counted as TP or FP. In the third dataset, ACTAINT identifies the init functions in these contracts as potential vulnerabilities. These may, in some cases, be preemptively initialized by others. In addition, some use modifiers based on `tx.origin`. Since the above issues do not appear in official reports, we consider them as potential vulnerabilities and do not count them as TPs.

VII. RELATED WORK

Many approaches have been proposed to detect vulnerabilities in smart contracts. Static analysis tools include Slither [24], [9], Mythril [38], [5], Securify [46], Maian [40], Oyente [36] and Smartcheck [45]. An earlier empirical study on 47,587 smart contracts [19] showed that static analysis tools are effective at detecting arithmetic and reentrancy vulnerabilities, but less accurate in identifying access control issues, denial of service, and front-running attacks.

Another category of effective approaches for detecting smart contract vulnerabilities is dynamic analysis, and one of the most effective techniques within this category is fuzzing. The earliest tool, ContractFuzzer [29], introduced instrumentation-based fuzzing. Later tools like ReGuard [33] and ILF [27] targeted specific vulnerabilities such as reentrancy and used policy-driven strategies. Over time, more advanced techniques emerged, including gray-box fuzzing [49], genetic algorithms [39], and data-flow-based feedback [18]. Tools like sfuzz [39], ContraMaster [47], and IR-Fuzz [35] improved test generation and seed optimization. More recent approaches, such as xFuzz [50] and MuFuzz [42], leverage machine learning and dynamic energy allocation to handle complex issues like cross-contract attacks and multi-function vulnerabilities.

Recently, several studies have explored using LLMs for smart contract vulnerability detection. Chen et al. [16] found that ChatGPT underperforms compared to traditional tools, especially in detecting access control issues. GPTLens [28] uses a multi-agent approach for general vulnerability detection, while GPTScan [43] targets logic vulnerabilities through rule-based preprocessing and post-validation, requiring substantial expert effort. To our knowledge, no existing LLM-based method specifically addresses access control vulnerabilities.

Several approaches have been proposed for detecting access control vulnerabilities, including TEETHER [31], Ethainter [14], Spcon [34], Somo [23] and AChecker [26]. Most of these methods rely on data flow and taint analysis to determine whether sensitive operations can be accessed by unauthorized entities. Given that AChecker demonstrates the highest performance among them, we adopt it as the baseline in our study.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we propose ACTAINT, an LLM-based multi-agent system for detecting access control vulnerabilities in smart contracts. Our method combines static analysis with the reasoning capabilities of large language models to enhance precision and interpretability. Specifically, we use static rules to guide the LLM in performing taint analysis within a two-agent framework. The sink agent identifies potential sinks in the contract based on the static rules and provides both the sink locations and corresponding reasoning explanations. The taint agent then takes the output of the sink agent and conducts data flow analysis to determine whether a taint path exists from a user-controlled input to the identified sink. If such a path exists, it indicates a potential access control vulnerability. To evaluate ACTAINT, we compare it with state-of-the-art static analysis detection tools, AChecker, by conducting experiments on CVE dataset and a large-scale real-world smart contract dataset. Overall, ACTAINT achieves a better performance.

In future work, we plan to incorporate more advanced static analysis techniques, such as symbolic execution, abstract interpretation and data flow analysis. These techniques can help us more precisely abstract critical program structures, which in turn can assist the LLMs in identifying vulnerabilities more effectively. Additionally, we intend to explore various fine-tuning strategies to enhance the LLMs' understanding of smart contract logic, aiming to reduce false positives and improve detection accuracy.

DATA AVAILABILITY

Our code and experimental results are available at [12]

ACKNOWLEDGMENT

This work is supported by the Zhejiang Pioneer (Jianbing) Project (2025C01198), the National Science Foundation of China (No.62372398, No.72342025), and the Fundamental Research Funds for the Central Universities (No.226-2025-00067). This work is also supported by the Open Research Fund of The State Key Laboratory of Blockchain and Data SecurityZhejiang University.

References

[1] Case 1 in false negative analysis. https://github.com/smartbugs/smartbugs-wild/blob/master/contracts/0xa82873dbb0835dca5c273363eeb006342e696036.sol.

[2] Case 2 in false negative analysis. https://github.com/smartbugs/smartbugs-wild/blob/master/contracts/0x0d5baa6b2bc5b62edf7ac263078fa4b9c5e9c040.sol.

[3] Case in false positive analysis. https://github.com/smartbugs/smartbugs-wild/blob/master/contracts/0x3ac0d29eaf16eb423e07387274a05a1e16a8472b.sol.

[4] Etherscan. https://etherscan.io.

[5] mythril github. https://github.com/Consensys/mythril.

[6] Parity wallet attack. https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html.

[7] Parity wallet attack. https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.

[8] The second motiviating example from smartbugs wild dataset. https://github.com/smartbugs/smartbugs-wild/blob/master/contracts/0x1981716911e621d725b835ba0af776e5f9be0bbe.sol.

[9] slither github. https://github.com/crytic/slither.

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

[11] solc-select. https://github.com/crytic/solc-select.

[12] Source code and log file. https://github.com/Smart-Contract-Security/ACTaint.

[13] Tn example. https://github.com/smartbugs/smartbugs-wild/blob/master/contracts/0x0f771aa18c5003aba1b0feee082a0dd6acd29956.sol.

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

[15] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.

[16] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Jianxing Yu, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. When ChatGPT Meets Smart Contract Vulnerability Detection: How Far Are We? *arXiv e-prints*, page arXiv:2309.05520, September 2023.

[17] Ruirui Chen, Chengwei Qin, Weifeng Jiang, and Dongkyu Choi. Is a large language model a good annotator for event extraction? In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17772–17780, 2024.

[18] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 227–239. IEEE, 2021.

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

[20] Lishui Fan, Mouxiang Chen, and Zhongxin Liu. Self-explained keywords empower large language models for code generation. *arXiv preprint arXiv:2410.15966*, 2024.

[21] Zezhong Fan, Xiaohan Li, Kaushiki Nag, Chenhao Fang, Topojoy Biswas, Jianpeng Xu, and Kannan Achan. Prompt optimizer of text-to-image diffusion models for abstract concept understanding. In *Companion Proceedings of the ACM Web Conference 2024*, pages 1530–1537, 2024.

[22] C Fang, X Li, Z Fan, J Xu, K Nag, et al. Llm-ensemble: optimal large language model ensemble method for e-commerce product attribute value extraction (2024). *arXiv preprint arXiv:2403.00863*.

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

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

[25] Menglin FU, Lifa WU, Zheng HONG, and Wenbo FENG. Research on vulnerability mining technique for smart contracts. *Journal of Computer Applications*, 39(7):1959, 2019.

[26] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. Achecker: Statically detecting smart contract access control vulnerabilities. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 945–956. IEEE, 2023.

[27] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 531–548, 2019.

[28] Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. Large language model-powered smart contract vulnerability detection: New perspectives. In *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 297–306. IEEE, 2023.

[29] Bo Jiang, Ye Liu, and Wing Kwong Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 259–269, 2018.

[30] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

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

[32] Junyi Li, Jie Chen, Ruiyang Ren, Xiaoxue Cheng, Wayne Xin Zhao, Jian-Yun Nie, and Ji-Rong Wen. The dawn after the dark: An empirical study on factuality hallucination in large language models. *arXiv preprint arXiv:2401.03205*, 2024.

[33] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 65–68, 2018.

[34] Ye Liu, Yi Li, Shang-Wei Lin, and Cyrille Artho. Finding permission bugs in smart contracts with role mining. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 716–727, 2022.

[35] Zhenguang Liu, Peng Qian, Jiaxu Yang, Lingfeng Liu, Xiaojun Xu, Qinming He, and Xiaosong Zhang. Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting. *IEEE Transactions on Information Forensics and Security*, 18:1237–1251, 2023.

[36] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.

[37] Daniel Macrinici, Cristian Cartofeanu, and Shang Gao. Smart contract applications within blockchain technology: A systematic mapping study. *Telematics and Informatics*, page 2337–2354, Dec 2018.

[38] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam*, 9(54):4–17, 2018.

[39] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020.

[40] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*, pages 653–663, 2018.

[41] Peng Qian, Zhenguang Liu, Xun Wang, Chen Jianhai, Bei Wang, and Zimmermann Roger. Digital resource rights confirmation and infringement tracking based on smart contracts. Jan 2019.

[42] Peng Qian, Hanjie Wu, Zeren Du, Turan Vural, Dazhong Rong, Zheng Cao, Lun Zhang, Yanbin Wang, Jianhai Chen, and Qinming He. Mufuzz: Sequence-aware mutation and seed mask guidance for blockchain smart contract fuzzing. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 1972–1985. IEEE, 2024.

[43] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. pages 1–13, 2024.

[44] Lingxiao Tang, Jiakun Liu, Zhongxin Liu, Xiaohu Yang, and Lingfeng Bao. Llm4szz: Enhancing szz algorithm with context-enhanced assessment on large language models. *arXiv preprint arXiv:2504.01404*, 2025.

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

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

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

[48] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[49] Valentin Wüstholz and Maria Christakis. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409, 2020.

[50] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 21(2):515–529, 2022.

[51] Lyuye Zhang, Kaixuan Li, Kairan Sun, Daoyuan Wu, Ye Liu, Haoye Tian, and Yang Liu. Acfix: Guiding llms with mined common rbac practices for context-aware repair of access control vulnerabilities in smart contracts. *arXiv preprint arXiv:2403.06838*, 2024.

[52] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. Blockchain challenges and opportunities: A survey. *International journal of web and grid services*, 14(4):352–375, 2018.