Have We Solved Access Control Vulnerability Detection in Smart Contracts? A Benchmark Study

Han Liu*, Daoyuan Wu†§, Yuqiang Sun‡, Shuai Wang*§, Yang Liu‡
*The Hong Kong University of Science and Technology, Hong Kong SAR, China
†Lingnan University, Hong Kong SAR, China
‡Nanyang Technological University, Singapore
Emails: liuhan@ust.hk, daoyuanwu@ln.edu.hk, suny0056@e.ntu.edu.sg, shuaiw@cse.ust.hk, yangliu@ntu.edu.sg

***Abstract*—Access control (AC) vulnerabilities are among the most critical security threats to smart contracts. Despite extensive research, they remain widespread and damaging in the Ethereum ecosystem. To understand and advance the current state-of-the-art (SOTA) in AC vulnerability detection, we first curate a diverse dataset of 180 real-world AC vulnerabilities from CVE entries, DeFiHackLabs incidents, and Code4rena audit reports.**

Using this dataset, we conduct a systematic benchmark study along three dimensions. First, we develop a cause-based taxonomy and analyze the prevalence and evolution of AC vulnerabilities. Second, we evaluate six SOTA tools, including two from industry and four from academia, revealing low recall (3% to 8%) and significant blind spots. To understand these failures, we examine 1.2 million deployed contracts and uncover practical gaps in AC protection mechanisms overlooked by existing tools. Finally, we assess the potential of large language models (LLMs) for AC vulnerability detection and show that LLMs detect 53–75% of vulnerabilities, outperforming traditional tools but facing challenges such as hallucinations and scalability. Our findings highlight the need for hybrid approaches that combine static analysis with LLM-based semantic reasoning to address the complexity of modern AC vulnerabilities.

***Index Terms*—Smart Contracts, Access Control, Vulnerability Detection, Large Language Models.**

I. Introduction

Smart contracts, self-executing programs deployed on blockchains, underpin a wide range of decentralized applications (dApps), particularly in decentralized finance (DeFi). Their transparency and programmability have driven rapid adoption, with DeFi protocols alone managing over $116 billion in total value locked (TVL) as of 2025 [1]. However, the immutability of smart contracts also introduces irreversible risks: a single vulnerability can result in catastrophic financial losses [2], [3]. A stark example is the 2025 multisig wallet phishing attack, which exploited a standard wallet implementation and caused a $1.4 billion loss [4].

Among these risks, access control (AC) vulnerabilities, failures in enforcing permissions on critical functions, pose one of the most severe threats. These vulnerabilities can allow unauthorized parties to steal funds, seize control of system components, or manipulate protocol behavior. AC vulnerabilities are ranked as the top security risk in the 2025 OWASP Top 10 for smart contracts [5], reflecting their prevalence and devastating impact. In 2024 alone, AC exploits caused $953.2 million in losses, accounting for 67.1% of the total $1.42 billion in damages from smart contract hacks [5]. This alarming trend has continued into 2025, exemplified by the $50 million exploit of the Infini protocol, in which an attacker gained unauthorized access to a privileged wallet.

§Corresponding authors: Daoyuan Wu and Shuai Wang.

TABLE I
Recent and severe incidents of AC vulnerabilities.

Target	Loss($)	Time	Target	Loss($)	Time
Infini	50M	2025-02-24	M2 Exchange	13.7M	2024-10-31
Orange Finance	0.83M	2025-01-08	Radiant Capital	53M	2024-10-16
98Token	0.03M	2025-01-04	Shezmu	4.9M	2024-09-20
Fegex	0.9M	2024-12-29	WXETA	0.11M	2024-09-16
BTC24H	0.09M	2024-12-16	Indodax	25.2M	2024-09-10

Table I lists several recent and severe AC-related exploits. These incidents reveal a persistent and systemic problem: despite continuous advances in smart contract auditing, AC vulnerabilities remain a dominant class of exploits. Attackers continue to exploit common design flaws such as missing access modifiers [6], flawed role assignments [7], and insecure cross-contract permission schemes [8]. Their increasing frequency and outsized financial impact underscore the urgency of developing robust and context-aware security analysis tools.

In response, researchers and practitioners have developed a range of vulnerability detection tools, including general-purpose static analyzers from industry, such as Slither [9] and Mythril [10], which use rule-based pattern matching and symbolic execution. Alongside these, specialized academic tools, such as AChecker [11], SPCon [12], PrettySmart [13], and SoMo [14], aim to detect AC-specific issues, including missing authorization and privilege escalations. Despite these efforts, real-world AC vulnerabilities remain prevalent, raising concerns about the practical effectiveness of existing tools.

Unfortunately, existing benchmark studies [15]–[20] have largely focused on general static application security testing (SAST) tools or specific vulnerability classes like reentrancy [20], leaving key limitations in the AC domain unaddressed. Specifically, three major gaps remain:

- *Lack of a real-world, diverse AC vulnerability dataset.* Prior studies evaluated tools on narrow or synthetic datasets, failing to capture the diversity of AC vulnerabilities observed in practice.
- *Absence of a unified, cause-based taxonomy.* Existing classifications (e.g., [19]) focus on rule-based patterns and do not reflect the broader semantic and contextual

challenges faced by AC-specific tools. For example, tools like PrettySmart [13] that target privilege escalations are not adequately represented in static rule taxonomies.
- *Limited understanding of tool performance on modern contracts.* Tool evaluations have primarily used small-scale or CVE-based benchmarks, which poorly reflect the complexity and modularity of contemporary smart contracts.

To bridge these gaps, we present the first comprehensive benchmark study focused specifically on AC vulnerability detection in smart contracts. We first construct a diverse dataset comprising 180 real-world AC vulnerabilities from three complementary sources: standardized CVEs [21], real-world exploits from DeFiHackLabs [22], and peer-reviewed audit findings from Code4rena [23]. This dataset significantly exceeds previous similar efforts (e.g., 34 reentrancy vulnerabilities in [20]) in both scale and diversity.

Using this dataset, we construct a cause-based taxonomy of AC vulnerabilities and analyze their characteristics and evolution over time (Section V-A). We then evaluate the detection capabilities of four SOTA AC-specific tools (SPCon, AChecker, PrettySmart, SoMo) and two general-purpose SAST tools (Slither, Mythril) against our dataset. The results reveal that all tools exhibit low recall (3%–8%) and fail to detect the majority of real-world AC vulnerabilities. To further understand these failures, we analyze 1.2 million verified smart contracts from Ethereum-compatible blockchains. Our results reveal that 54.68% of AC protections, especially `this`-based and hybrid mechanisms, are poorly supported or completely missed by existing tools (Section V-C).

Finally, we explore the use of large language models (LLMs) in AC vulnerability detection. We test two general-purpose models (GPT-4o-mini, GPT-4o) and two reasoning-enhanced models (GPT-o3-mini, DeepSeek-R1) using our dataset. Our results show that these LLMs achieve substantially higher recall (53–75%) than traditional tools, demonstrating strong semantic reasoning capabilities. However, they also suffer from hallucinations and scalability limitations. These trade-offs highlight the need for hybrid detection frameworks that combine the strengths of both static analysis and LLM-based reasoning. Our study provides actionable insights into the limitations of current AC vulnerability detection methods and highlights promising directions for future research. All datasets and evaluation code are released on our repository [24].

In summary, this paper makes the following contributions:
- We curate a real-world, diverse dataset of 180 AC vulnerabilities from CVEs, DeFiHackLabs incidents, and Code4rena audit reports, and construct a cause-based taxonomy to analyze their characteristics and evolution.
- We systematically evaluate four SOTA AC-specific tools and two general-purpose SAST tools, revealing low recall and identifying critical gaps in AC vulnerability detection.
- We explore the potential of LLMs in detecting AC vulnerabilities, demonstrating their semantic strengths, identifying their operational trade-offs, and informing future hybrid detection strategies.

II. Preliminary

A. Access Control Vulnerabilities

AC vulnerabilities occur when systems fail to enforce restrictions on who or what can interact with resources, data, or functionalities [25]. In traditional software, these vulnerabilities often stem from misconfigured role-based access control (RBAC), privilege escalation flaws, or insufficient validation of user permissions [26]. For example, a web application might expose an administrative API without proper authentication checks, allowing attackers to manipulate critical settings. In smart contracts, AC mechanisms are programmatically enforced, introducing unique challenges [12]. Unlike traditional systems, due to their immutable nature, flawed AC logic in smart contracts cannot be patched without upgrades, which may introduce new risks. For instance, missing modifier checks (e.g., onlyOwner) or improper role assignments in smart contracts can grant unauthorized actors control over funds or governance mechanisms, resulting in significant financial losses. The urgency of addressing AC vulnerabilities in smart contracts is underscored by industry reports such as the OWASP Top 10 for Smart Contracts 2025, which identifies AC failures as the top one security risk in decentralized applications [5].

B. Definition of AC Vulnerabilities

Based on the concept of AC vulnerabilities [25] in traditional software, we formally define an AC vulnerability in smart contracts as follows:

Let a smart contract be modeled as a state transition system represented by the tuple:

$$\mathcal{C} = (S, F, A, P, \delta)$$

where:
- S is the set of all possible states of the contract.
- F is the set of functions in the contract
- A is the set of actors (e.g., owned accounts, contracts).
- $P : F \to 2^A$ maps functions to authorized actor sets.
- $\delta : F \times A \times S \to S$ is the state transition function.

An **access control vulnerability** exists iff:

$$\exists f \in F,\, a \in A,\, s \in S \quad \text{s.t.} \quad a \notin P(f) \wedge \delta(f, a, s) \neq \bot$$

where $\bot$ represents an invalid or blocked transition.

III. Overview

In this section, we present an overview of our study, which aims to answer the following research questions (RQs):

RQ1: Taxonomy of AC Vulnerabilities. What are the dominant types of real-world AC vulnerabilities in smart contracts, and how has their prevalence evolved over time?

RQ2: Effectiveness of SOTA Tools. How effective are SOTA tools in detecting real-world multi-sourced AC vulnerabilities, and what are their key limitations?

RQ3: Practical Gaps. How do current AC protection mechanisms in smart contracts function, and why do existing tools fail to detect the absence of these protections?

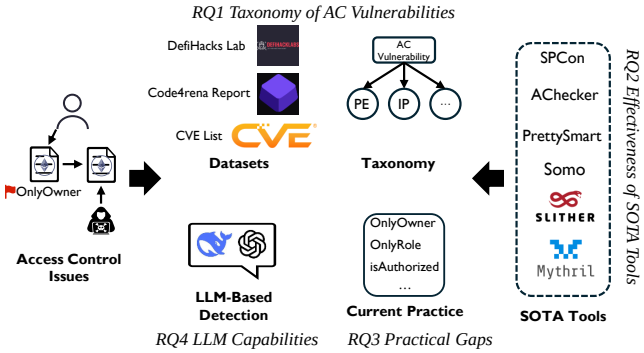

Fig. 1. The framework of our AC vulnerability study.

RQ4: LLM Capabilities. Can LLMs complement traditional tools to improve AC vulnerability detection?

Fig. 1 illustrates the overall framework of our study, which systematically addresses the four RQs through four interconnected phases: data curation, tool evaluation, practical validation, and LLMs' integration. First, we establish a multi-source dataset of real-world AC vulnerabilities to analyze their cause-based taxonomy and temporal evolution, directly addressing RQ1. Then, leveraging this dataset, we evaluate the effectiveness of SOTA detection tools (e.g., Slither, AChecker) in our dataset, addressing RQ2 by quantifying recall, and blind spots. Thirdly, we investigate practical gaps by contrasting the tools' focus with AC protection mechanisms in deployed contracts, answering RQ3. Finally, we explore the potential of LLMs (e.g., GPT-4o) to complement traditional tools, testing their ability to detect semantic vulnerabilities (e.g., lack of AC protection) under well-designed prompting strategies, thereby addressing RQ4.

Next, we present the study's process and methodology in Section IV, followed by the results in Section V.

IV. Study Design

In this section, we introduce the design of our study, which includes dataset and taxonomy construction (Section IV-A), evaluation tool selection (Section IV-B), current practice mining for practice gap analysis (Section IV-C), and an LLM-based detection method to explores the potential of LLMs in detecting AC vulnerabilities (Section IV-D).

A. Dataset and Taxonomy Construction

1) Dataset Construction: To systematically investigate AC vulnerabilities in smart contracts, we construct a multi-source dataset capturing both formally disclosed and real-world exploits. It integrates three complementary sources: CVE entries, DeFiHackLabs incidents, and Code4rena audit reports, ensuring comprehensive coverage across diverse contexts.

We start with the CVE database, a standardized cybersecurity repository, extracting all smart contracts entries since 2017 [21]. Two authors independently review these entries to identify AC vulnerabilities based on the definition in Section II-B. Disagreements are resolved through discussion with a third author, resulting in 21 AC vulnerabilities. However, since 2021, the CVEs have primarily documented limited smart contract vulnerabilities, mostly within OpenZeppelin codebases, leading to coverage gaps with modern smart contracts.

To address this limitation, we supplement CVE data with incidents from DeFiHackLabs [22], a repository of real-world smart contract attacks. We manually review all entries from 2017 to 2024, retaining cases where the incidents' root causes are satisfied with the definition of AC vulnerabilities. For closed-source contracts, we include only those with available bytecode to enable bytecode-level analysis. This yields 73 AC vulnerabilities, including high-impact exploits like unauthorized burn in the `Safemoon` contract [27], resulting in a total economic loss of $367.2 million.

To further enhance coverage, we incorporate Code4rena audit reports [23], which provide peer-reviewed, collective analyses of smart contract code. We examine all reports from January 2021 to December 2024, focusing on findings of "medium" severity or higher to prioritize actionable risks while excluding lower-severity issues like gas optimizations. Two authors independently reviewed each report to identify AC vulnerabilities based on the definition with disagreements resolved through discussion. This phase contributes 86 AC vulnerabilities to the dataset, including complex issues like insecure cross-contract permissions.

Finally, we harmonize the dataset by removing duplicates, resulting in 180 unique AC vulnerabilities: 21 from CVE, 73 from DeFiHackLabs' incident, and 86 from Code4rena reports, including 18 closed-source cases (bytecode only). In total, the dataset comprises 165 vulnerable contracts and 180 vulnerable functions, spanning over 380,095 lines of code. The dataset captures a diverse range of AC issues, from simple modifier misconfigurations to complex cross-contract authorization flaws, reflecting the severity and variety of real-world issues.

2) Taxonomy Development: To systematically analyze the AC vulnerabilities in our dataset, we need a taxonomy based on their root causes. This taxonomy serves as a foundation for understanding the nature of AC vulnerabilities and their evolution over time, directly addressing RQ1. While foundational works like the Common Weakness Enumeration (CWE) [28] and Li *et al.*'s SAST-oriented taxonomy [19] provide a classification standard, neither suffices for AC-specific analysis. Li *et al.*.'s taxonomy defines 18 SAST-friendly subtypes (e.g., `UnprotectedEtherWithdrawal`), but its narrow focus on static code patterns excludes most of the AC vulnerabilities in our dataset. Similarly, CWE's general-purpose categories lack the granularity to address blockchain-specific attack vectors like signature-based AC and focus more on traditional software instead of smart contracts.

However, CWE's concept provides a useful starting point for our taxonomy development. Consequently, drawing on CWE-284 [29], we analyze the dataset to identify the types of AC vulnerabilities employing open card sorting [30], a qualitative research technique used to identify patterns and classify items based on shared characteristics. Initially, each vulnerability was written on an individual card. Two authors independently review the cards and categorize them into different types according to the root causes. After the initial categorization, the authors

discuss the categorization results and resolve any discrepancies through consensus. The final taxonomy is reviewed by all authors to ensure consistency and completeness. Ultimately, we identify five primary types of AC vulnerabilities in smart contracts, details are presented in Section V-A.

B. Tool Selection

To obtain the current state of AC vulnerability detection tools, we conduct a literature review targeting academic publications from top-tier conferences in software engineering and cybersecurity, including ICSE, FSE, ASE, ISSTA, OOPSLA, USENIX Security, IEEE S&P, CCS, and NDSS, between 2020 and 2024. Using search terms of "smart contract", we identify 146 papers, which are manually reviewed to determine whether they proposed tools explicitly designed for AC vulnerability identification. This process reveal three dedicated AC detection tools: SPCon [12], which analyzes historical transaction data and leveraging a role mining algorithm to effectively identify permission bugs; AChecker [11], a static analyzer applies static data flow analysis to examine the bytecode of smart contracts for AC vulnerabilities; and PrettySmart [13], which targets detecting privilege escalation vulnerabilities, by inferring permission constraints. Additionally, we include SoMo [14], a tool aimed at detecting modifier bypass vulnerabilities.

To ensure the evaluation's generalization of modern smart contract ecosystems, we extend our analysis to general-purpose SAST tools. Following the criterion outlined in [19], we exclude tools that were no longer actively maintained or lacked compatibility with Solidity versions post-0.8.0, as outdated tools often fail to parse newer language features or optimize for recent practices. After applying these criteria, two widely adopted SAST tools remain: Slither, a highly configurable static analysis tool that supports custom rules for AC-specific checks, and Mythril, a symbolic execution engine capable of detecting AC vulnerabilities through path exploration.

C. Current Practice Mining

To explore the gap between theoretical vulnerability detection and real-world developer practices, we conduct a large-scale study of AC mechanisms in deployed smart contracts. Our analysis focuses on 1,286,847 unique verified contracts across Ethereum-compatible blockchains [31], encompassing 5,148,625 deployments sourced from EtherScan, BscScan, BaseScan, etc. This dataset represents a comprehensive snapshot of AC practices in decentralized applications (dApps), DeFi protocols, etc., providing insights into how developers implement and enforce AC in practice.

We systematically extract and analyze three types of validation constructs: `require` statements, `if-revert` blocks, and `assert` statements, which are the primary mechanisms for enforcing AC logic in Solidity. For each contract, we parse its source code to identify all instances of these constructs, recording their conditional expressions and error messages. To ensure consistency, we normalize the logic of if-revert blocks by logically inverting their conditions (e.g., if (msg.sender != owner) revert(); is transformed to require(msg.sender == owner)). This normalization enables direct comparison with require and revert patterns. To ensure a focused analysis, we exclude conditions that occur fewer than 50 times, retaining only frequently recurring patterns for manual review. In this phase, we first assess whether the conditions are related to AC mechanisms. Relevant conditions are then categorized into different types using open card sorting [30], as shown in Section IV-A2. We also investigate the gap between tool detection and real-world AC mechanisms, detailed in Section V-C4.

Prompt for LLM-based AC Vulnerabilities Detection

Now you are a **smart contracts security audit expert**, you are now doing audit on some smart contracts to find access control issues in it. You need to find all the possible access control issues in the given file of the smart contracts. You **first** need to analyze the context in which the contract operates. Understand the variables and functions that need to be restricted in each specific context. **Then**, analyze each state variable and function in sequence. If you discover that a public function fails to perform the necessary access control checks before invoking certain functions or modifying certain variables, this constitutes a potential access control issue. Based on this, you need to create a proof of concept to verify it. Please finally out put the vulnerable function name, line and the reasons in the response.
For example, the contract is:
<CONTRACT>

```
1  contract Ownable {
2      address public owner;
3      function Ownable() public {
4          owner = msg.sender;
5      }
6      modifier onlyOwner() {
7          require(msg.sender == owner);
8          _;
9      }
10     function transferOwnership(address
           newOwner) onlyOwner public {
11         require(newOwner != address(0));
12         OwnershipTransferred(owner, newOwner
               );
13         owner = newOwner;
14     }
15     function withdraw() onlyOwner public {
16         uint256 etherBalance = address(this)
               .balance;
17         owner.transfer(etherBalance);
18     }
19 }
```

</CONTRACT>
Output:
1. Function Ownable() (line 4-6) has the access control issues. Reason: The Ownable() function can change the owner variable which is significant because with the role, we can do anything on the contract.

Fig. 2. The prompt for AC vulnerability detection.

D. LLM-based Detection Method

In this study, we extend AC vulnerability detection to LLMs, benchmarking their capabilities against traditional tools and

exploring their potential for enhancing detection. We design a structured prompting framework that integrates role-based task specification, chain-of-thought (CoT) reasoning [32], and few-shot learning. Four models are evaluated: two general-purpose LLMs (GPT-4o-mini, GPT-4o) and two reasoning-enhanced models (GPT-o3-mini, DeepSeek-R1).

The LLMs are assigned the role of a smart contract security audit expert, with explicit instructions to identify AC vulnerabilities through systematic analysis. The prompt begins with a task definition: "Now you are a smart contracts security audit expert, you are now doing audit on some smart contracts to find AC issues in them." This role-based framing primes the models to adopt an expert mindset, emphasizing the background knowledge and reasoning skills required for effective detection. To emulate human-like reasoning, we employ the CoT approach, breaking down the audit into sequential steps. First, the models are instructed to analyze the contract's operational context, identifying variables and functions requiring restrictions. Next, they are directed to examine each state variable and function, flagging public functions lacking necessary AC checks as potential vulnerabilities. Finally, they need to generate a proof of concept to verify the issue, ensuring practical relevance. We also employ a simple few-shot learning strategy to enhance detection accuracy. The prompt includes examples of AC vulnerabilities with corresponding detection rationales, allowing the models to learn from explicit instances. The full prompt is shown in Fig. 2.

V. Study Results

A. RQ1: Taxonomy of AC Vulnerabilities

In this RQ, we aim to identify the dominant types of real-world AC vulnerabilities in smart contracts and analyze their prevalence over time. We first categorize 180 real-world AC vulnerabilities into five root cause-based subtypes, including *lack of AC protection*, *privilege escalation*, *insufficient validation*, *incorrect AC validation*, and *signature-related issues*. The distribution of these subtypes is shown in Fig. 3.

Lack of AC Protection (LAC): This type refers to the complete absence of access restrictions on sensitive functions or state variables. These vulnerabilities arise when developers neglect to implement modifiers (e.g., onlyRole) or encapsulate critical variables. For instance, a Code4rena audit of `Maia DAO` reveals an unprotected `payableCall` function, allowing anyone to steal all non-native assets (ERC20 tokens, NFTs, etc.) [33]. This subtype dominates the dataset (60%, 108/180), with 72% of Code4rena findings attributed to such oversights, a trend linked to rushed development cycles and incomplete security reviews in fast-moving DeFi ecosystems.

Privilege Escalation (PE): It occurs when a function does have ACs but contains logic flaws that let unauthorized actors (or authorized ones) exploit it to gain elevated privileges. A classic example is CVE-2018-10705 [34], where a `IDXM Token` contract's `setOwner` function lacks caller validation, enabling arbitrary role assignments. The main difference between this category and LAC is that PE leverages flawed or incomplete ACs to illegally elevate privileges, whereas LAC entirely lacks

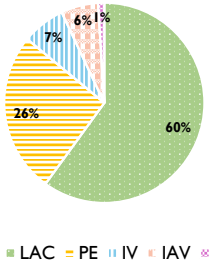

Fig. 3. The distribution of AC types in our datasets.

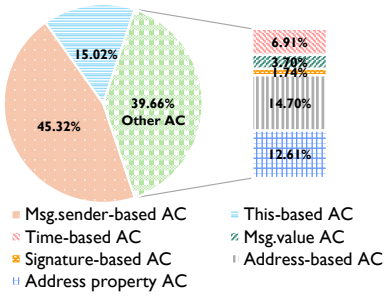

Fig. 4. The distribution of the AC mechanisms in 1.2 million smart contracts.

access restrictions on sensitive functions or state variables. While PE accounts for 26% (47/180) of cases overall, it is historically prevalent in early CVEs (about 80% of pre-2020 entries). Modern PE cases also exist due to the complexity of DeFi protocols and their interactions across multiple contracts.

Insufficient Validation (IV): This category describes partial or incomplete AC-related checks that fail to account for all attack vectors. For example, a `LiquidXv2Zap` contract's `deposit` function does not check that the `account` should be `msg.sender`, thus account's approval on the `zap` can be spent to buy tokens and add liquidity [35]. Such vulnerabilities represent 7% of cases, often emerging in systems integrating multiple protocols.

Incorrect AC Validation (IAV): This category encompasses logically flawed or misconfigured checks, such as using incorrect variables or overly permissive conditions. A 2024 Code4rena audit identifies a vulnerability in the function `initializeAlphaIndices()` of contract `EntropyGenerator`, where the function is intended to be called by the `TraitForgeNft` contract. However, it is currently protected by the `onlyOwner` modifier. This means only the owner of the `EntropyGenerator` contract can call it, not the `TraitForgeNft` contract [36]. This category accounts for 6% of cases, often stemming from complex contract interactions and logical errors in AC checks or improper use of modifiers. All these cases are occurred in Code4rena audit reports due to the complex contract interaction in audited contracts.

Signature-Related Issues (SR): It involves cryptographic validation flaws in signed message workflows. These include replay attacks due to missing nonces (e.g., a vesting contract accepting expired signatures) and weak signature schemes like using ecrecover without address uniqueness checks. For instance, a vulnerability in the `BGeoToken` contract allows attackers to input empty data into `_r`, `_s` and `_v` to bypass signature checks to mint tokens [37]. This category accounts for 1% of cases, often stemming from improper use of cryptographic or logical errors in signature verification.

> **Finding 1:** Our analysis categorizes 180 real-world AC vulnerabilities into five root cause-based subtypes: *lack of AC protection*, *privilege escalation*, *insufficient validation*, *incorrect AC validation*, and *signature-related issues*.

As for the time evolution of AC vulnerabilities, we observe that, in the early years of smart contract adoption (2017–2019), PE dominated the vulnerability landscape, accounting for

about 80% of CVE entries during this period. This prevalence stems from junior developer awareness and the absence of standardized libraries. For example, both CVE-2018-10666 [38] and CVE-2018-10705 [34] are PE vulnerabilities caused by bypassing owner validation through the `setOwner` function. In contrast, LAC vulnerabilities have become the most common issues in recent years (2020–2025), accounting for 72% of Code4rena findings. This shift reflects the rapid growth of DeFi ecosystems and the increasing complexity of smart contract interactions, which have outpaced developers' ability to implement robust AC, e.g., `Maia DAO` contract [33] mentioned above. Concurrently, SR and IAV vulnerabilities began appearing in audits, reflecting the growing adoption of cryptographic primitives and novel governance mechanisms. Source-specific trends further highlight the difference in these vulnerabilities. CVEs and the incidents recorded in DeFiHackLabs, which have more PE vulnerabilities (37% of cases), typically represent "late-stage" vulnerabilities, flaws discovered after deployment, such as the incident in contract `LinearVesting`, where the `init` function can set the message sender as the owner to bypass `onlyOwner` check [39]. In contrast, Code4rena audits, which focus on pre-deployment reviews, predominantly identify LAC vulnerabilities (72% of findings), such as public `lockOnBehalf` function in `LockManager` contracts to repeatedly extend a user's unlockTime [40].

> **Finding 2:** As audited contracts mitigate basic AC flaws (e.g., missing AC), attackers now exploit architectural complexity, targeting weak cross-contract validation and cryptographic gaps—areas where traditional methods lag.

B. RQ2: Effectiveness of SOTA Tools

1) Experimental Setup: All the experiments were conducted on a server with an AMD Ryzen Threadripper 3970X 32-Core Processor and 252 GB of RAM. For Slither and Mythril, we follow the settings in the [17]. For other tools, we utilize their open-source code. We set the time limit to 30 minutes, tripling the baseline values within their respective publications [11]–[14].

2) Results and Analysis: We conduct function-level root cause alignment, comparing the root causes of warnings reported by tools against the ground-truths in our dataset. All the results have been manually reviewed by two authors to ensure correctness. The results are shown in Table II.

Our evaluation of six SOTA tools reveals critical gaps in AC vulnerability detection. Slither, a static analyzer, reported 8 TPs but suffered severe false positives (389 FPs), primarily in DeFiHackLabs and Code4rena datasets. Mythril, a symbolic execution engine, detected only 4 TPs and timed out in 25.9% of cases (42/162). Specialized tools showed narrow utility: SPCon identified 9 TPs in CVEs but failed on Code4rena pre-deployment audits due to its reliance on historical data. AChecker achieved 12 CVEs TPs but collapsed in real-world scenarios (0 TPs). PrettySmart detected 13 TPs on CVEs but missed all cases (74 FNs) in Code4rena, while SoMo focuses on modifier misuse, yielding 13 TPs but ignoring 64% of vulnerabilities (104 FNs) involving non-modifier AC checks.

> **Finding 3:** All the tools failed to detect a significant number of AC vulnerabilities, with the most one only detecting 8% vulnerabilities. Slither reported the most FPs (389), while other tools exhibited few or no FPs.

We further analyzed the reasons for the missing detections and FPs to identify common blind spots. The limited effectiveness of Slither in detecting AC vulnerabilities arises from rigid pattern-based rules and a narrow set of AC-specific detection rules. With only 10 predefined AC rules, the tool struggles to address the diversity and complexity of vulnerabilities observed in practice. For example, the *arbitrary-send-eth* rule [41] checks for unprotected Ether transfers but misses vulnerabilities like PE, where attackers bypass modifiers through indirect paths. This pattern-matching approach also fails to handle dynamic or context-dependent permissions, e.g., role-based systems distributed across multiple contracts. Additionally, the inability of Slither to track cross-contract interactions or inherited logic results in false negatives. In Code4rena audits, 84 false negatives occurred because Slither could not recognize AC mechanisms distributed across modular contracts, such as governance logic delegated to separate modules.

As for FPs, Slither over-relies on keyword matching in these rules and overlooks contextual protections. For example, its *controlled-delegatecall* rule [42] might flag a secure `delegatecall` to a fixed/immutable address (e.g., an upgradeable proxy pattern) or one guarded by strict access controls (e.g., `onlyOwner`), simply because the target is technically a user-provided parameter, ignoring the context of the call.

The limitations of Mythril stem from its design as a symbolic execution engine and its constrained rule set. With only 8 predefined detection patterns, Mythril also struggles to cover the diverse range of AC vulnerabilities present in different contracts. For example, while it includes rules for generic vulnerabilities like *dependence_on_tx_origin* [43], it lacks specialized logic to identify PE or IAV, leading to 116 false negatives across datasets. The computational demands of symbolic execution further exacerbate these issues. Mythril explores all possible execution paths to detect vulnerabilities, a method that theoretically offers precision but becomes impractical for some modern, complex contracts. For instance, analyzing a contract with recursive call logic could trigger path explosion, exhausting computational resources and resulting in failed cases (27% of total evaluations) due to timeouts, even with a 24-hour runtime limit (setting suggested in [17]). While symbolic execution reduces false positives (0 FPs observed), its exhaustive approach inadvertently suppresses valid warnings, as many true vulnerabilities reside in paths that Mythril cannot feasibly explore within the time constraints. Additionally, Mythril also struggles to model context-specific permissions, such as cross-contract interactions or dynamic role assignments. For example, a governance contract where permissions depend on voting outcomes would evade detection because Mythril cannot simulate such states.

As for the specific tools, SPCon, AChecker, PrettySmart, and SoMo, we found that SPCon is heavily dependent on the availability of historical transaction data, which restricts its

TABLE II
EFFECTIVENESS OF SOTA TOOLS AND LLMS CAPABILITIES ON DETECTING AC VULNERABILITIES (CVE MEANS COMMON VULNERABILITIES AND EXPOSURES SOURCE, DF MEANS DEFIHACKLABS SOURCE, C4 MEANS CODE4RENA SOURCE).

	True Positive (TP)				False Positive (FP)				False Negative (FN)				Failed			
	CVE	DF	C4	Overall	CVE	DF	C4	Overall	CVE	DF	C4	Overall	CVE	DF	C4	Overall
Slither	1	6	1	8	10	198	181	389	20	49	84	153	0	0	1	1
Mythril	4	0	0	4	0	0	0	0	7	42	67	116	10	13	19	42
SPCon	9	1	NA	10	0	0	NA	0	10	40	NA	50	2	14	NA	16
AChecker	12	0	0	12	0	0	0	0	8	19	38	65	1	54	48	103
PrettySmart	13	0	0	13	0	0	1	1	8	61	74	143	0	12	12	24
SoMo	10	2	1	13	0	0	0	0	11	47	46	104	0	6	39	45
GPT-4o-mini	17	39	65	121	69	345	537	951	4	16	21	41	0	0	0	0
GPT-4o	18	36	49	103	56	189	243	488	3	19	37	59	0	0	0	0
GPT-o3-mini	19	36	39	94	1	23	78	102	2	19	47	68	0	0	0	0
DeepSeek-R1	16	33	37	86	7	30	25	62	5	22	49	76	0	0	0	0

applicability to pre-deployment or newly deployed contracts. For example, in Code4rena audits, where contracts are analyzed before deployment and thus have no transaction history, SPCon produced no results because its role-mining algorithm relies on analyzing past interactions to infer permissions. Even when transaction data exists, SPCon fails to detect vulnerabilities in functions that have never been invoked or cannot be reached through bypassing existing call paths. For instance, a privileged `burn` function in a `DxBurnToken` contract [44] remains undetected because no transactions have ever triggered it. This reliance on historical activity creates blind spots for latent vulnerabilities that exist in code but have not yet been used, limiting SPCon's utility in proactive security audits.

AChecker faces three core limitations in this task. First, it struggles with cross-contract AC, due to its bytecode-level intra-contract analysis. The second limitation is its failure to infer implicit AC requirements tied to critical state variables. For example, in the `cERC20` contract [6], the `mint` function (shown in Fig. 5) modifies security-sensitive variables like `_totalSupply` and `_balances` (line 7 and line 8) but lacks explicit permission checks. AChecker missed this vulnerability because it focuses on syntactic patterns (i.e., msg.sender validation) rather than recognizing that state transitions on `_totalSupply` inherently require AC. Third, AChecker's reliance on symbolic execution results in significant performance bottlenecks. Its exhaustive path exploration often leads to timeouts, particularly in large or recursive codebases.

As for PrettySmart, its detection capabilities are constrained by its reliance on taint analysis and a predefined set of permission constraints (PCs) tied to msg.sender validation. The tool infers AC policies by tracking five types of permission constraints. However, this approach mirrors shortcomings of AChecker in handling implicit or non-msg.sender-based permissions. Similarly, in the Fig. 5, PrettySmart failed to detect the unprotected `_totalSupply` and `_balances` modifications because these operations lack direct msg.sender checks or modifiers, leading to false negatives. Additionally, while PrettySmart avoids the scalability issues of symbolic execution (i.e., no timeout failures), its taint analysis introduces trade-offs. The tool cannot reason about complex conditional logic or contextual permissions. For instance, a function guarded by a hybrid check like `require(block.timestamp > unlockTime || msg.sender == admin)` might be misclassified as secure

```
1  function mint(...) public returns (bool) {
2    _mint(acccount, amount);  //public mint
3    return true;
4  }
5  function _mint(...) internal {
6    require(account != address(0), "...");
7    _totalSupply = _totalSupply.add(amount);
8    _balances[...] = _balances[...].add(...);
9    emit Transfer(address(0), account, amount);
10 }
```

Fig. 5. A vulnerable `mint` function in a `cERC20` contract.

if the taint engine cannot track the relationship between block.timestamp and administrative privileges, resulting in false positives or negatives.

SoMo's narrow focus on modifier bypass issues severely limits its applicability to broader AC challenges. The tool is designed to detect scenarios where attackers circumvent modifiers (e.g., calling a function without triggering its onlyOwner modifier), but it ignores other critical AC flaws, i.e., without a modifier. In our dataset, some vulnerabilities involved non-modifier issues, including insecure cross-contract calls and role misconfigurations. SoMo failed to detect these cases entirely, as its design does not account for them. This specialization renders SoMo ineffective in modern ecosystems where AC logic increasingly relies on hybrid mechanisms.

> **Finding 4:** General-purpose tools struggle to detect AC vulnerabilities due to rigid rule sets, while specialized tools exhibit narrower strengths with significant limitations in cross and complex contracts and implicit permissions.

C. RQ3: Practical Gaps

In RQ2, we analyze the weaknesses of various tools, finding that both AChecker and PrettySmart primarily detect AC issues based on conditions related to `msg.sender`. Building on this, we now investigate the AC protection mechanisms implemented in existing smart contracts, assessing whether they are adequately covered by current analysis tools. Following the methodology described in Section IV-C, we found that 21.31% of these conditions pertained to AC mechanisms, which we categorize into three groups: *msg.sender-based AC mechanisms*, *this-based AC mechanisms*, and *other AC mechanisms*. The following subsections provide a detailed taxonomy of these AC mechanisms, and the distribution of them is shown in Fig. 4.

1) Msg.sender-based AC Mechanisms: AC mechanisms center on msg.sender validation dominate real-world smart

contracts, reflecting their foundational role in permission management. The most prevalent approach involves comparing msg.sender and a predefined address (35.16% of observed cases), where functions validate msg.sender against hardcoded administrative roles like owner, admin, or controller. For example, a DeFi protocol might restrict critical operations such as adjusting interest rates to a designated admin address using `require(msg.sender == admin)`. Beyond comparing to a predefined address, data-driven AC (3.07%) tie permissions to on-chain states or dynamic conditions. A common example is ERC20 allowance checks, where transferFrom functions validate `require(allowance[owner][msg.sender] >= amount)` to enforce delegated spending limits. This approach enables fine-grained control over permissions by linking access rights to real-time data, such as token balances or voting outcomes.

Another key pattern is operator-based AC (2.72%). Here, third-party addresses approved by primary owners gain limited privileges. For instance, an ERC721 contract might allow operators to transfer tokens on behalf of owners by `require(isApprovedForAll(owner ,msg.sender))`, enabling delegated management without full ownership rights. Similarly, role-based AC (2.10%) assigns privileges through roles rather than predefined addresses, as seen in contracts using OpenZeppelin's AccessControl library. A token minting function might enforce `require(hasRole(MINTER_ROLE, msg.sender))`, restricting minting to authorized roles.

Group-based AC (1.36%) enforces permissions through predefined whitelists or other group lists stored in mappings. For example, a contract might maintain a whitelist mapping and validate callers with `require(whitelist[msg.sender])`, restricting functions like token minting or governance participation to approved addresses. This mechanism allows administrators to dynamically update permissions by adding or removing addresses from the list, offering flexibility for complex systems like membership-based platforms. Finally, some contracts implement address validation (0.92%) by querying external states or contracts. A notable example is NFT-gated access, where a function verifies if msg.sender owns a specific NFT, such as `require(ownerOf(tokenId) == msg.sender)` in a private club contract.

> **Finding 5:** The diversity in msg.sender-based protections, including comparing msg.sender and a predefined address AC, querying-based AC, operator-based AC, role-based AC, group-based AC, and data-driven AC, underscores the importance of context-aware AC in real-world contracts.

2) This-based AC Mechanisms: AC mechanisms leveraging `this` (the current contract instance) are less common but vital for securing internal or contract-initiated actions, accounting for 15.02% of observed AC conditions. These mechanisms primarily enforce permissions by validating that interactions originate from the contract itself or depend on its internal state.

The most common pattern in this category, data-driven AC (14.24%), ties permissions to the contract's runtime state. For example, a liquidity pool might restrict withdrawals until its balance meets a threshold using `require(address(this).balance >= MIN_RESERVE)`. A smaller but critical subset involves comparing `this` address with a predefined address (0.7%), where functions restrict execution to the contract itself. For example, a contract might manage ownership transitions by `require(owner == address(this))`, ensuring that only internal logic (e.g., a governance vote) can update ownership. This prevents external actors from directly modifying critical parameters, requiring changes to follow predefined workflows.

Another pattern employs query-based AC (0.01%), where the contract validates its ownership of external assets to authorize actions. A licensing system, for instance, might use `require(nft. ownerOf(tokenId) == address(this))` to verify that the contract holds a specific NFT license before enabling privileged operations. This ensures that only contracts possessing the required NFT can execute sensitive functions, tying AC to verifiable on-chain assets. A third mechanism is operator-based delegation (0.06%). A contract might use `operatorFilterRegistry.isOperatorAllowed( address(this), operator)` to check if the operator is in the whitelist for specific tasks, such as managing contract funds or executing administrative functions. This approach allows contracts to delegate permissions to external entities while maintaining internal oversight over delegated actions.

> **Finding 6:** This-based AC mechanisms, including data-driven AC, comparing this address with predefined address AC, query-based AC, and operator-based AC, secure smart contracts by enforcing predefined permissions for internal actions and state transitions.

3) Other AC Mechanisms: Beyond `msg.sender` and `this`-based AC, smart contracts employ other AC strategies that defy traditional detection methods, accounting for 39.66% of observed AC conditions. These mechanisms often blend multiple validation criteria or rely on external data, creating challenges for existing tools.

The most frequent pattern is address-based AC (14.70%). It enforces permissions through address checks, similar to msg.sender-based AC but with other predefined addresses. For example, a staking contract might enforce a separation of roles by `require(owner != operator)`, ensuring that the contract owner and delegated operator cannot be the same entity. This prevents conflicts of interest, such as an operator abusing privileges if they also hold ownership rights. Another prevalent pattern is address property AC (12.61%), which enforces permissions based on address characteristics. For example, a contract might block smart contracts from participating in a token sale via `require(isContract(target))` to prevent bot interference.

Apart from these, time-based AC (6.91%) restricts access based on temporal conditions. For instance, a vesting contract might restrict withdrawals until a predefined timestamp using `require(block. timestamp >= unlockTime)`, ensuring funds remain locked until maturity. Msg.value AC

(3.70%) regulates access through payment requirements. A premium feature in a dApp might require users to pay a fee using `require(msg.value >= 0.1 ether)`, granting access only to paying addresses. Signature-based AC represents another critical category (1.74%), where permissions are granted via cryptographic proofs. For instance, a cross-chain bridge might require users to submit signed messages from authorized validators, verified using `ecrecover`.

> **Finding 7:** Smart contracts also utilize diverse AC mechanisms (i.e., address-based, time-based, msg.value, and signature-based) beyond basic `msg.sender` and `this` checks. These mechanisms create context-dependent permission rules that challenge automated detection tools.

4) Summary of Findings: Our findings of AC mechanisms in real-world smart contracts reveal several practical gaps that hinder existing tools' ability to detect vulnerabilities. First, tools primarily focus on `msg.sender`-based permissions, overlooking other critical mechanisms like `this`-based or other AC. For example, AChecker and PrettySmart, which rely on `msg.sender` patterns, would miss vulnerabilities tied to `this` or contract state conditions. Second, tools struggle with complex or hybrid permissions that combine multiple validation criteria. For instance, Slither and Mythril, which target simple `msg.sender` patterns would fail to detect time-based or signature-based permissions. Third, tools lack semantic context awareness, leading to false positives or negatives in nuanced AC scenarios. For example, AChecker might misclassify a data-driven permission as secure if it cannot infer the relationship between contract state and access rights.

D. RQ4: LLM Capabilities

Building on findings from RQ2 and RQ3, which revealed that traditional tools struggle to detect vulnerabilities like LAC due to their inability to infer semantic context (e.g., critical state variables), in this RQ, we explore the potential of LLMs to address these gaps.

1) Experimental Setup: Following the methodology in Section IV-D, we evaluate four LLMs: GPT-4o-mini-2024-07-18, GPT-4o-2024-08-06, GPT o3-mini-2025-01-31, and DeepSeek-R1 hosted on Microsoft Azure. Temperature settings were fixed at 0 for all models except o3-mini, which does not support temperature adjustments, to minimize output randomness and reproduction. For the CVE and DeFiHackLabs datasets, we analyze the entire codebase, while for the Code4rena datasets, we focus our analysis on the vulnerable contracts identified in the reports due to the substantial size of these projects. We believe this approach does not significantly affect detection performance and results, as the vulnerable contracts generally represent the most critical components within the audits.

2) Results and Analysis: Similar to RQ2, we manually review the outputs of LLMs to identify TPs and FPs, the results are shown in the Table II. GPT-4o-mini achieved the highest recall (74.7%, 121 TPs) but also the highest FPs (951), outperforming the best traditional tool SoMo (8.0% recall). GPT-4o balanced precision and recall better (103 TPs, 488 FPs), while GPT-o3-mini prioritized precision (94 TPs, 102 FPs). DeepSeek-R1 showed moderate results (86 TPs, 55 FPs).

TABLE III
THE RESULT OF LLMS ON DETECTING AC VULNERABILITIES WITHOUT DATA LEAKAGE.

	TP	FP	FN	Total Cases
GPT-4o-mini	40	304	15	55
GPT-4o	29	155	26	55
GPT-o3-mini	25	47	30	55
DeepSeek-R1	4	10	8	12

```
1 modifier onlyMinter() {
2     msg.sender == minterAddress; // No revert
3     _;
4 }
```

Fig. 6. A vulnerable `onlyMinter` modifier from Code4rena reports.

To mitigate the impact of data leakage on detection, we also evaluated LLMs on a sanitized dataset, excluding the vulnerability reported before training (knowledge cutoff dates: GPTs: 2023-10, DeepSeek-R1: 2024-07). As shown in Table III, while TP declined across models, FPs remained stable, and overall performance still surpassed traditional tools (13 TPs).

Overall, LLMs demonstrate unique strengths in detecting AC vulnerabilities by leveraging their semantic understanding of code context, akin to human auditors. For example, all evaluated LLMs successfully identified a critical flaw in a Code4rena audit report where a modifier [45] lacked a revert statement (line 2) (shown in Fig. 6). Traditional tools like Slither or AChecker missed this issue because they focus on their syntactic patterns, e.g., PE or modifier bypass, rather than logical correctness, highlighting the ability of LLMs to infer intent from incomplete patterns.

However, LLMs also exhibit limitations. Their performance notably declines when lacking explicit context. They often struggle with complex, multi-layered vulnerabilities involving cross-contract interactions or indirect state changes. In the `addLiquidity` function [46] shown in Fig. 7, the assignment `listToken[_token]=true` (line 3) introduced a flaw by enabling unauthorized token listings, but only DeepSeek-R1 detected this. Other models produced FPs. This reflects LLMs' limited ability to track implicit state dependencies across contracts or distinguish between intentional and hazardous patterns. In contrast, specialized tools like AChecker or PrettySmart, which focus on specific AC patterns, might have detected this vulnerability if they recognized the criticality of the `listToken` state variable.

In addition, LLMs' propensity for hallucination leads to FPs in cases where protections exist but are overlooked. For instance, in the function `setUPs` [47] shown in Fig. 8, GPT-4o, GPT-4o-mini, and GPT-o3-mini erroneously flagged this as a vulnerability, failing to recognize the required check in line 2. In contrast, DeepSeek-R1 avoided this FP by reasoning.

```
1 function addLiquidity(...) public {
2   ...;
3   listToken[_token] = true; // AC flaw
4   users[_token][...].tz += 100 ether;
5 }
```

Fig. 7. A vulnerable `addLiquidity` function in DeFiHackLabs's incidents.

```
1 function setUPs(...) public {
2   require(upaddress[to] == msg.sender); // AC
3   upaddress[to] = newAddr;
4 }
```

Fig. 8. A non-vulnerable `setUPs` function with AC mechanism.

TABLE IV
THE RESULT OF LLMS ON DETECTING AC VULNERABILITIES WITH AN ALTERNATIVE PROMPT.

	TP	FP	FN	Total Cases
GPT-4o-mini	106	516	56	162
GPT-4o	91	358	71	162
GPT-o3-mini	79	102	83	162
DeepSeek-R1	85	126	77	162

This highlights the trade-off between precision and recall in LLMs, where high recall may lead to increased false positives due to overfitting on common patterns. Models with enhanced reasoning capabilities, such as DeepSeek-R1, reduce FPs by contextualizing code semantics but at the cost of lower TP rates. This trade-off underscores the challenge of balancing precision and recall in LLM-based detection.

In order to further analyze the potential of LLMs in detecting AC vulnerabilities, we conducted additional experiments using a new prompt that assigns LLM the role of "an experienced smart contract developer" with optimized instructions for clarity and structured reasoning. The results are presented in Table IV. Overall, with the alternative prompt, both TPs and FPs decreased modestly. The LLM became slightly more conservative in its judgments. However, the overall effectiveness remained similar: the order of magnitude of FPs was still significant, and simply modifying the prompt did not fundamentally improve precision or recall. This further confirms that combining LLMs with SA or other hybrid methods is necessary to achieve practical accuracy.

> **Finding 8:** LLMs excel at identifying explicit, contextually obvious vulnerabilities (e.g., missing modifiers) effectively but struggle with complex AC logic. Hybrid approaches combining LLMs and static analysis could address multi-contract workflows while reducing hallucinations via iterative prompts or fine-tuning.

Cost Analysis. To explore the practicality of deploying LLMs, we analyzed both financial and temporal costs. Running all models on per-vulnerability detection costs of $0.018, $0.016, $0.001, and $0.004, respectively. The GPT models delivered results rapidly, with response times of 5–10 seconds per analysis, while DeepSeek-R1 required about 2 minutes per query due to computational optimizations. Despite variations in speed and cost, all models operated within economically and temporally acceptable thresholds, making them viable options for enhancing AC vulnerability detection.

VI. DISCUSSION

A. Implications for Practitioners

Our study provides practical implications for both smart contract developers and tool creators. For *developers*, we show that widely-used static analyzers often fail to detect access control vulnerabilities that rely on implicit logic or cross-contract context. We recommend supplementing these tools with manual code reviews and dynamic analysis, especially for contracts with complex permission structures. Developers should also be cautious about over-relying on modifiers, and instead adopt hybrid AC patterns, e.g., combining role checks, time constraints, and signature validations. These patterns are harder to bypass but are currently under-supported by tools.

For *tool creators*, our findings reveal critical blind spots in current analyzers stemming from their reliance on shallow syntactic rules and lack of semantic understanding. To improve AC vulnerability detection, future tools should: (1) broaden rule coverage to include overlooked AC enforcement styles (e.g., using `tx.origin`, contract ownership via `this`, or multi-signature patterns); (2) incorporate inter-contract reasoning to handle delegation and proxy contracts, which are increasingly prevalent; and (3) improve alert quality by providing contextual explanations and ranking results based on risk and confidence.

Our results also show that LLMs are capable of identifying semantic and context-dependent flaws that escape conventional tools. While LLMs alone are not yet production-ready due to scalability and hallucination issues, integrating them into static analyzers as semantic engines or pre-filters offers a promising hybrid path forward. These improvements can help bridge the gap between academic prototypes and the practical needs.

B. Future Directions

Our work in RQ4 focused on whether general LLM models have the potential to uncover AC issues missed by traditional static analysis tools. As our findings show, pure LLM-based approaches can do so, but still face significant challenges with hallucinations and incur additional costs. To address these limitations, future research could explore hybrid frameworks that combine LLMs' semantic reasoning with static analysis tools' syntactic checks. For instance, LLMs could be used to pre-screen contracts and identify high-risk functions for deeper analysis by traditional tools, or to provide contextual explanations for alerts generated by static analyzers, thereby improving their interpretability and reducing false positives. Another promising direction is to finetune LLMs with domain-specific knowledge from smart contract security, which could enhance their understanding of AC patterns and reduce hallucinations. This could involve training on a curated dataset of verified vulnerabilities and secure coding practices, enabling the models to learn more accurate representations of AC logic.

C. Threats to Validity

1) External Validity: A key threat to validity is the size of the vulnerability dataset, which could bias results toward specific AC flaw types. To mitigate this, we aggregate data from CVE entries, incidents recorded in DeFiHackLabs, and Code4rena audit reports, ensuring diversity across disclosure sources and vulnerability patterns. Another concern is the representativeness of contracts in practice mining, as analyzing a non-comprehensive sample might skew findings. We address this by including 1.2 million verified contracts from

Ethereum-compatible chains, covering DeFi protocols, NFTs, governance systems, etc. Finally, LLM's data leakage could inflate performance metrics if models encountered test samples during training. To minimize this risk, we evaluated LLMs on a sanitized validation dataset, excluding potential data leakage.

2) Internal Validity: Internal validity risks include bias during data collection and classification, where subjective judgments in labeling could introduce inconsistencies. To minimize this, two authors independently classified each vulnerability using open card sorting, with further disagreements resolved through consensus discussions. Another potential limitation stemmed from tool execution failures. To mitigate this, we adhered to default settings from their respective publications [11]–[14] for all tools (Slither and Mythril are following the settings in [17]), tripled the time limit to 30 minutes (vs. 10 minutes in their respective publications [11]–[14]), and retried failed executions to maximize valid outputs. Additionally, LLM's configuration variability might impact detection consistency. We mitigated this by fixing LLM temperatures to 0, ensuring reproducibility and reducing noise in outputs.

VII. Related Work

Empirical Study on Vulnerability Detection for Smart Contracts. Several studies have evaluated vulnerability detection tools for smart contracts. Rameder *et al.* [48] conducted a literature review on security tools utilizing fuzzing, formal methods, and static analysis. Ghaleb *et al.* [15] proposed SolidiFI to inject 9,369 vulnerabilities, evaluating six static analysis tools' detection capabilities. Durieux *et al.* [16] assessed tools on 69 labeled and 47,518 unlabeled contracts, revealing accuracy limitations in real-world settings. Ren *et al.* [17] proposed a four-step methodology to assess nine security tools on 46,186 contracts. Monteiro [18] developed SmartBugs, a framework evaluating seven tools on 47,661 contracts for scalability. Li *et al.* [19] introduced a fine-grained vulnerability taxonomy and evaluated eight SAST tools via an extensible benchmark. Chaliasos *et al.* [49] evaluated 5 SOTA security tools on 127 attacks and conducted a survey of 49 developers and auditors. Their finding emphasize the need to develop specialized tools catering to the distinct demands, which also motivate our work.

Several studies have also examined dynamic security tools. For example, Wu *et al.* [50] conducted an empirical evaluation of the usability and effectiveness of fuzzing-based tools. In addition, related work has also focused on constructing benchmarks for tool evaluation. Zheng *et al.* [20] developed a benchmark dataset specifically targeting reentrancy-related security issues. Furthermore, Zheng *et al.* [51] utilized the Smart Contract Weakness Classification Registry to build datasets based on 1,199 security audits.

Our research differs fundamentally from the aforementioned studies in several key aspects: (*i*) We focus exclusively on a single domain, AC vulnerabilities, which have emerged as one of the most severe security risks with distinct demands in smart contracts in recent years. (*ii*) We categorize a cause-based taxonomy of AC vulnerabilities and conduct a detailed evaluation and analysis of SOTA security tools in this domain. (*iii*) We explore the potential of LLMs in detecting AC issues, providing valuable insights and directions for future research.

Access Control Vulnerability Detection and Repair for Smart Contracts. AC vulnerabilities are critical threats to smart contracts, driving diverse detection and mitigation solutions. Liu *et al.* [12] introduced SPCon, the first tool for detecting permission-related issues by analyzing historical transactions. Ghaleb *et al.* [11] proposed AChecker, which applies static data flow analysis to detect AC vulnerabilities. Zhong *et al.* [13] developed PrettySmart, targeting privilege escalation by inferring permission constraints. Fang *et al.* [14] introduced SoMo to identify AC issues related to bypassing modifiers. Apart from these specialized tools, general SAST tools also employ predefined security rules to identify AC vulnerabilities, including Slither [9], Mythril [10], Securify [52], Manticore [53], Osiris [54], Oyente [55], and Maian [56]. Beyond detection, Zhang *et al.* [57] proposed ACFix, which mines common RBAC to repair AC issues using LLMs.

VIII. Conclusion

This study presents a comprehensive investigation into AC vulnerability detection in smart contracts. We constructed a multi-source dataset of 180 real-world AC vulnerabilities, enabling a cause-based taxonomy and temporal evolution analysis. Our systematic evaluation of SOTA tools revealed critical limitations: existing approaches achieved only 3–8% recall due to insufficient handling of semantic complexity and context-dependent permissions. In parallel, we explored the potential of LLMs, which demonstrated superior semantic reasoning (53–75% recall) but faced challenges in hallucination mitigation and scalability. By integrating empirical gap analysis, longitudinal vulnerability trends, and LLM capabilities, this paper underscores the urgent need for adaptive, context-aware detection frameworks. These findings provide actionable insights for advancing AC security in smart contract ecosystems.

Acknowledgements

We thank all reviewers for their constructive comments. This research is partially supported by a research fund provided by HSBC, HKUST TLIP Grant FF612, and Lingnan Grant SUG-002/2526. This research is also supported by the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG4-GC-2023-008-1B); by the National Research Foundation Singapore and the Cyber Security Agency under the National Cybersecurity R&D Programme (NCRP25-P04-TAICeN); and by the Prime Minister's Office, Singapore under the Campus for Research Excellence and Technological Enterprise (CREATE) Programme. Any opinions, findings and conclusions, or recommendations expressed in these materials are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, Cyber Security Agency of Singapore, Singapore.

REFERENCES

[1] "Defillama," https://defillama.com/, May 2025.

[2] H. Liu, D. Wu, Y. Sun, H. Wang, K. Li, Y. Liu, and Y. Chen, "Using my functions should follow my checks: Understanding and detecting insecure OpenZeppelin code in smart contracts," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 3585–3601. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/liu-han

[3] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639117

[4] "Defillama hacks," https://defillama.com/hacks, May 2025.

[5] "Owasp smart contract top 10," https://owasp.org/www-project-smart-contract-top-10/, May 2025.

[6] "Public mint," https://web3sec.notion.site/c582b99cd7a84be48d972ca2126a2a1f?v=4671590619bd4b2ab16a15256e4fbba1&p=c26e99d8419a4fca930183100b0a32f6&pm=s, May 2025.

[7] "Public transferownership," https://web3sec.notion.site/c582b99cd7a84be48d972ca2126a2a1f?v=4671590619bd4b2ab16a15256e4fbba1&p=ea529b8b0b77424691035bee47044c2a&pm=s, May 2025.

[8] "Benddao incident," https://code4rena.com/reports/2024-07-benddao, May 2025.

[9] J. Feist, G. Grieco, and A. Groce, "Slither Analyzer," Jun. 2023. [Online]. Available: https://github.com/crytic/slither

[10] "Mythril," https://github.com/Consensys/mythril, May 2025.

[11] A. Ghaleb, J. Rubin, and K. Pattabiraman, "Achecker: Statically detecting smart contract access control vulnerabilities," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 945–956.

[12] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 716–727.

[13] Z. Zhong, Z. Zheng, H.-N. Dai, Q. Xue, J. Chen, and Y. Nan, "Prettysmart: Detecting permission re-delegation vulnerability for token behaviors in smart contracts," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024.

[14] Y. Fang, D. Wu, X. Yi, S. Wang, Y. Chen, M. Chen, Y. Liu, and L. 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*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1157–1168. [Online]. Available: https://doi.org/10.1145/3597926.3598125

[15] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 415–427.

[16] T. Durieux, J. a. F. Ferreira, R. Abreu, and P. 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*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 530–541.

[17] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, "Empirical evaluation of smart contract testing: what is the best choice?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 566–579.

[18] A. P. C. Monteiro, "A study of static analysis tools for ethereum smart contracts," Ph.D. dissertation, Master's thesis, Instituto Superior Técnico, 2019.

[19] K. Li, Y. Xue, S. Chen, H. Liu, K. Sun, M. Hu, H. Wang, Y. Liu, and Y. Chen, "Static application security testing (sast) tools for smart contracts: How far are we?" *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1447–1470, 2024.

[20] Z. Zheng, N. Zhang, J. Su, Z. Zhong, M. Ye, and J. Chen, "Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 295–306.

[21] "Common vulnerabilities and exposures," https://www.cve.org/CVERecord/SearchResults?query=smart+contract, May 2025.

[22] "Past defi incidents," https://github.com/SunWeb3Sec/DeFiHackLabs, May 2025.

[23] "Code4rena audits reports," https://code4rena.com/reports, May 2025.

[24] "Study data of the paper," 2025. [Online]. Available: https://github.com/HelayLiu/AccessControlVulnerabilities

[25] M. Dalton, C. Kozyrakis, and N. Zeldovich, "Nemesis: Preventing authentication & [and] access control vulnerabilities in web applications," in *18th USENIX Security Symposium (USENIX Security 09)*. USENIX Association, 2019.

[26] F. Sun, L. Xu, and Z. Su, "Static detection of access control vulnerabilities in web applications," in *20th USENIX Security Symposium (USENIX Security 11)*. San Francisco, CA: USENIX Association, Aug. 2011.

[27] "Safemoon incident," https://web3sec.notion.site/c582b99cd7a84be48d972ca2126a2a1f?v=4671590619bd4b2ab16a15256e4fbba1&p=fa3c3cffd3254c0181ae3f9cd640890b&pm=s, May 2025.

[28] "Common weakness enumeration," https://cwe.mitre.org/, May 2025.

[29] "Cwe-284: Improper access control," https://cwe.mitre.org/data/definitions/284.html, May 2025.

[30] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[31] "Sourcify," https://github.com/ethereum/sourcify, May 2025.

[32] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[33] "Maia dao incident," https://code4rena.com/reports/2023-09-maia, May 2025.

[34] "Cve-2018-10705," https://www.cve.org/CVERecord?id=CVE-2018-10705, May 2025.

[35] "Liquidxv2zap incident," https://web3sec.notion.site/c582b99cd7a84be48d972ca2126a2a1f?v=4671590619bd4b2ab16a15256e4fbba1&p=b703b4b9bf134e40a65f716f87277602&pm=s, May 2025.

[36] "Entropygenerator incident," https://code4rena.com/reports/2024-07-traitforge, May 2025.

[37] "Bego token incident," https://web3sec.notion.site/c582b99cd7a84be48d972ca2126a2a1f?v=4671590619bd4b2ab16a15256e4fbba1&p=5f0b5f9f1412498fb569f538c4e8ce88&pm=s, May 2025.

[38] "Cve-2018-10666," https://www.cve.org/CVERecord?id=CVE-2018-10666, May 2025.

[39] "Linearvesting incident," https://web3sec.notion.site/c582b99cd7a84be48d972ca2126a2a1f?v=4671590619bd4b2ab16a15256e4fbba1&p=31b06970c5584811a942b3835b7b052f&pm=s, May 2025.

[40] "Lockmanager incident," https://code4rena.com/reports/2024-05-munchables, May 2025.

[41] "Arbitrary send eth," https://github.com/crytic/slither/blob/master/slither/detectors/functions/arbitrary_send_eth.py, May 2025.

[42] "Controlled delegatecall," https://github.com/crytic/slither/blob/master/slither/detectors/statements/controlled_delegatecall.py, May 2025.

[43] "Dependence on origin," https://github.com/Consensys/mythril/blob/develop/mythril/analysis/module/modules/dependence_on_origin.py, May 2025.

[44] "Public burn," https://web3sec.notion.site/c582b99cd7a84be48d972ca2126a2a1f?v=4671590619bd4b2ab16a15256e4fbba1&p=f91fe4168c59469fb5f5deefc9e00af0&pm=s, May 2025.

[45] "Rabbithole incident," https://code4rena.com/reports/2023-01-rabbithole, May 2025.

[46] "Addliquidity incident," https://web3sec.notion.site/c582b99cd7a84be48d972ca2126a2a1f?v=4671590619bd4b2ab16a15256e4fbba1&p=c30d3fce9e13476ea116137aefcf3a10&pm=s, May 2025.

[47] "Stakingrewards contract," https://bscscan.com/address/0x274b3e185c9c8f4ddef79cb9a8dc0d94f73a7675, May 2025.

[48] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Ethereum smart contract analysis tools: A systematic review," *IEEE Access*, vol. 10, pp. 57 037–57 062, 2022.

[49] S. Chaliasos, M. A. Charalambous, L. Zhou, R. Galanopoulou, A. Gervais, D. Mitropoulos, and B. Livshits, "Smart contract and defi security tools: Do they meet the needs of practitioners?" in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser.

ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024.

[50] S. Wu, Z. Li, L. Yan, W. Chen, M. Jiang, C. Wang, X. Luo, and H. Zhou, "Are we there yet? unraveling the state-of-the-art smart contract fuzzers," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2024.

[51] 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 Transactions on Software Engineering*, vol. 50, no. 6, pp. 1360–1373, 2024.

[52] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 67–82.

[53] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[54] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th annual computer security applications conference*, 2018, pp. 664–676.

[55] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[56] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 653–663.

[57] L. Zhang, K. Li, K. Sun, D. Wu, Y. Liu, H. Tian, and Y. Liu, "Acfix: Guiding llms with mined common rbac practices for context-aware repair of access control vulnerabilities in smart contracts," *IEEE Transactions on Software Engineering*, 2025.