ACFix: Guiding LLMs With Mined Common RBAC Practices for Context-Aware Repair of Access Control Vulnerabilities in Smart Contracts

Lyuye Zhang, Kaixuan Li, *Graduate Student Member, IEEE*, Kairan Sun, Daoyuan Wu, *Member, IEEE*, Ye Liu, Haoye Tian, and Yang Liu, *Senior Member, IEEE*

***Abstract*—Smart contracts are susceptible to various security issues, among which access control (AC) vulnerabilities are particularly critical. While existing research has proposed multiple detection tools, automatic and appropriate repair of AC vulnerabilities in smart contracts remains a challenge. Unlike commonly supported vulnerability types by existing repair tools, such as reentrancy, which are usually fixed by template-based approaches, the main obstacle of repairing AC vulnerabilities lies in identifying the appropriate roles or permissions amid a long list of non-AC-related source code to generate proper patch code, a task that demands human-level intelligence. In this paper, we employ the state-of-the-art GPT-4 model and enhance it with a novel approach called ACFix. The key insight is that we can mine common AC practices for major categories of code functionality and use them to guide LLMs in fixing code with similar functionality. To this end, ACFix involves offline and online phases. In the offline phase, ACFix mines a taxonomy of common Role-based Access Control practices from 344,251 on-chain contracts, categorizing 49 role-permission pairs from the top 1,000 unique samples. In the online phase, ACFix tracks AC-related elements across the contract and uses this context information along with a Chain-of-Thought pipeline to guide LLMs in identifying the most appropriate role-permission pair for the subject contract and subsequently generating a suitable patch. To evaluate ACFix, we built the first benchmark dataset of 118 real-world AC vulnerabilities, and our evaluation revealed that ACFix successfully repaired 94.92% of them, a major improvement compared to the baseline GPT-4 at only 52.54%. We also conducted a human study to understand the value of ACFix's repairs and their differences from human repairs.**

***Index Terms*—Smart contract, software security, program repair.**

Received 22 October 2024; revised 29 June 2025; accepted 4 July 2025. This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 1 (RG96/23). It is also supported by the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-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. Recommended for acceptance by M. R. Lyu. *(Lyuye Zhang and Kaixuan Li contributed equally to this work.) (Corresponding author: Daoyuan Wu.)*

Lyuye Zhang, Kaixuan Li, Kairan Sun, and Yang Liu are with the College of Computing and Data Science, Nanyang Technological University, Singapore 639798.

Daoyuan Wu is with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong SAR 999077, China. He is now with the School of Data Science, Lingnan University, Hong Kong SAR 999077, China (e-mail: daoyuanwu@ln.edu.hk).

Ye Liu is with the Singapore Management University, Singapore, 188065.

Haoye Tian is with the University of Luxembourg, 9999 luxembourg, luxembourg.

Digital Object Identifier 10.1109/TSE.2025.3590108

I. Introduction

Smart contracts, Turing-complete programs executed on blockchain ledgers, implement predefined programmatic logic through transaction-based invocation [1]. With the emergence of decentralized applications such as DeFi [2] and NFTs [3], the use of smart contracts, especially those written in Solidity [4] on the Ethereum blockchain [1], has significantly expanded within the blockchain ecosystem. Nevertheless, these contracts can be susceptible to various security vulnerabilities, including reentrancy [5], integer overflow [6], front-running [7], price manipulation [8], etc. Among these, Access Control (AC) vulnerabilities [9] are particularly critical because they directly expose privileged operations to attackers, such as taking over the ownership of the contract or minting more tokens, which often lead to tremendous financial loss, e.g. an infamous attack, Parity [10].

Considering the severe implications associated with access control (AC) vulnerabilities, several automated detection tools have been recently introduced to mitigate these risks, such as Ethainter [11], SPCon [12], AChecker [9], and SoMo [13]. Among these tools, SPCon distinguishes itself by analyzing historical transactions to infer AC policies. In contrast, the other approaches primarily employ taint analysis techniques to trace critical instructions (e.g., `selfdestruct`) or state variables (e.g., `owner`), thereby identifying potential scenarios where unauthorized parties might gain access. While these works have thoroughly addressed the detection of AC vulnerabilities, they have not provided concrete guidance or recommendations on how to remediate these issues. Furthermore, although the tools can identify potential vulnerabilities effectively, they lack an explainable reasoning process to justify or clarify the rationale behind their detections.

While detecting AC vulnerabilities has certain information flow patterns, repairing them needs a step further to identify appropriate roles or permissions. As a result, although numerous repair tools for smart contracts have been proposed [14], [15], [16], [17], [18], [19], [20], only a few of them support AC vulnerability repairs. Unfortunately, although certain repair systems—such as Elysium [19] and SmartFix [14]—explicitly claim support for addressing access control (AC) vulnerabilities, their scope is restricted to fixing only a limited set of common unauthorized operations, such as *Re-initialization* [21], *Suicidal* [22], and *Low-level Call* [23]. However, these predefined AC misuse patterns are insufficient to comprehensively cover the complexity and diversity encountered in real-world smart contract implementations, leading to the inability of successful patch generation.

Existing vulnerability repair tools are often constrained by predefined access control restrictions specific to the contract's owner, rendering template-based repair approaches potentially adequate for standard operational scenarios. However, the current methodological landscape presents a significant limitation in addressing unauthorized privilege escalation across broader contexts, particularly in more complex AC vulnerabilities that require nuanced automated repair mechanisms. For instance, the motivating example presented in §II illustrates that an unprotected `deposit` function can also lead to unforeseen financial losses for smart contracts. This privilege should be granted to the role `Bank` rather than the contract's owner for more flexibility.

In general, automatically and appropriately repairing AC vulnerabilities in smart contracts requires human-level intelligence. This is because AC policies in smart contracts are commonly enforced through the Role-Based Access Control (RBAC) [24] mechanism, which requires setting appropriate RBAC *roles* that align with corresponding privileged operations (referred to as *permissions* in RBAC terminology). Intuitively, for a repair system to function effectively, it must *(i)* first achieve a human-level understanding of the functionality embedded within the vulnerable code, *(ii)* then recognize appropriate RBAC roles based on this understanding, and *(iii)* finally generate correct patches. Although recent advancements in large language models (LLMs) [25], [26] allow us to utilize state-of-the-art (SOTA) models like GPT-4 [26], accomplishing these three tasks still presents challenges.

Specifically, For **task *(i)***, determining AC-related operations from the raw code corpus is even hard for GPT-4, given the substantial noise present within the source code. Compounding this challenge, LLMs are known to have limited attention spans, leading to a loss of focus [27]. To address this issue, we have developed a static slicing algorithm to extract the relevant code context, allowing GPT-4 to focus on it. For **task *(ii)***, off-the-shelf LLMs were not inherently trained to recognize RBAC roles and their typical privileged operations, i.e., the mapping of role-permission pairs. Moreover, LLM hallucination [28] could lead to unreliable output. Hence, it becomes essential to build an RBAC taxonomy, derived from common RBAC practices in smart contracts, for the LLM to select from. For **task *(iii)***, the patches generated might conflict with pre-existing, inaccurately implemented RBAC mechanisms. Therefore, besides building new RBAC from scratch, we also mine existing RBAC mechanisms from the source code and reuse them in the generated patches. Our evaluation suggests that this strategy is effective for addressing inadequately implemented RBAC. Another issue for task *(iii)* is that LLMs' randomness could still occasionally divert the LLM from generating correct patches. To address this, we implemented a Multi-Agent Debate (MAD) mechanism [29] to establish a loop between *generator* and *validator*. With such validation, *validator* can effectively suppress *generator*'s hallucination and ensure the generation of proper patches.

Based on the observations above, we propose a novel approach named ACFIX to enhance the capabilities of the state-of-the-art GPT-4 model in repairing AC vulnerabilities in smart contracts. The key insight is that we can mine common AC practices from major categories of code functionality and use these practices to guide LLMs in fixing code with similar functionality. Specifically, ACFIX first conducts offline mining of common RBAC practices from 344,251 on-chain contracts and builds an RBAC taxonomy consisting of 49 role-permission pairs from the top 1,000 pairs mined. ACFIX then utilizes the mined common RBAC practices as a "knowledge base for AC repair" to guide LLMs in fixing code with similar functionality. To help LLMs understand the functionality of the vulnerable code, ACFIX employs static code slicing to extract AC-related code context, more specifically, an AC context graph (ACG). With this two-fold source of information, ACFIX instructs GPT-4 to follow the Chain-of-Thought (CoT) [30] prompting to identify the proper role-permission pairs. Eventually, ACFIX generates the patch and validates it according to the original vulnerability description.

We conducted evaluations comparing ACFIX with SOTA tools [14], [15] and performed an ablation study to highlight the improvements of individual components ACFIX offers over the baseline GPT-4. To comprehensively evaluate repair tools, we collected and constructed a benchmark dataset consisting of 118 cases from real-world attacks and contracts. To the best of our knowledge, this is the first benchmark dataset specifically for AC vulnerabilities. Our results showed that ACFIX successfully repaired 94.92% of AC vulnerabilities using appropriate AC mechanisms. The ablation study further revealed that without the enriched context and mined taxonomy supplied by ACFIX, vanilla GPT-4 fixed 52.54% of vulnerabilities. *validator* agent further boosted the fixing rate from 87.28% to 94.92%. Additionally, we analyzed the repair capabilities of tools across various role-permission pairs by category as well as their monetary and time costs.

Furthermore, to understand the value of ACFIX's repairs and how they differ from human repairs, we conducted a human-based evaluation involving 10 experts who have worked on smart contract auditing for 2-7 years. The results show that ACFIX's repairs are mostly aligned with those of humans and are even finer-grained than those of both senior and junior experts, although in rare cases (3/118), human experts are better at handling open issues based on their knowledge and experience without much guidance. Moreover, around half of the AC fixes are non-trivial to devise by humans, indicating that ACFIX can provide a unique complement to assist human-in-the-loop repair as a copilot.

Contributions. To sum up, our contributions are as follows:

- We proposed ACFIX, the first tool designed to repair AC vulnerabilities by guiding LLMs to appropriately enforce RBAC mechanisms across a variety of scenarios.
- We assembled the first benchmark dataset of 118 AC vulnerabilities, sourced from real-world attacks and contracts, based on which, we conducted an extensive evaluation of the effectiveness and efficiency of ACFIX and SOTA tools and LLMs, including an ablation study.
- We obtained a taxonomy of common RBAC practices, including 49 role-permission pairs summarized from the top 1K unique samples mined from 344,251 on-chain contracts.
- We carried out a human study to understand the value of ACFIX's repairs, yielding new insights into the comparison between LLM-based and human repairs.

II. Background and Motivation

A. Background

Large Language Model. Pre-trained language models such as BERT [31] and GPT [32] have revolutionized the field of natural language processing (NLP) through pre-training on large text corpora. This approach has enabled these models to develop robust, transferable language representations that are highly effective across a wide range of NLP applications. Based on our evaluation of four popular LLMs, including GPT-4 [33], GPT-3.5 [34], Mistral [35], and Llama3 [36], in §VII-A, we eventually use GPT-4 as the base mode.

Smart Contract. Smart contracts are self-executing agreements where the terms are encapsulated in executable code and run on blockchains [37]. However, smart contracts may be susceptible to software vulnerabilities, leading to financial risks. If a contract allows for unauthorized ERC20 [38] token transfers, a flaw like improper access control can expose it to risks such as malicious abuse of legitimate functions.

Role-based Access Control (RBAC) [24] is a well-known security paradigm in which *permissions* are assigned to *roles* rather than directly to users. Each user belongs to one or more roles to accomplish various access control policies. This approach encapsulates a set of permissions within each role, defining the actions a user can perform. Nowadays, RBAC is recommended as the state-of-the-art security practice for separating the execution of access control policies from the management of business logic in smart contracts, usually through a set of well-defined modifiers [13], [39].

B. A Motivating Example

Our approach was motivated by a real-world AC attack on the DeFi application named *GYMNetwork* [40], [41]. Fig. 1 shows the vulnerable function `depositFromOtherContract`, the root cause of which is that it is marked as `external`. Without the validation by an appropriate modifier, an attacker was able to deposit numerous fake tokens to falsify his token shares in *GYMNetwork*, leading to a loss of two million USD in 2022.

The patch provided by the original author added a modifier, `onlyBank`, to ensure that only the vault address can deposit

```
function depositFromOtherContract(uint256
    _depositAmount, uint8 _periodId,
    bool isUnlocked, address _from
) external  { //vulnerable, fixed by onlyBank
    require(isPoolActive,'Not running yet');
    _autoDeposit(_depositAmount,_periodId,
        isUnlocked,_from);
}
```

Fig. 1. An example of smart contract AC vulnerabilities.

tokens. Since the role `Bank` had already been defined in the vulnerable contract, RBAC was partially implemented by the author previously. In this case, the vulnerable function could have been repaired with existing RBAC mechanisms from the code context, by `onlyBank`, in accordance with the *plastic surgery hypothesis* [42]. If the context is not considered during the repair, existing tools, such as SmartFix [14], and LLMs (GPT-4) adopted conservative measures, i.e. `owner of the contract`, as in §VII-C, which could lead to overfitting by inappropriately preventing legitimate banks from depositing. Clearly, this not the expected behavior, as such repairs significantly impede the function's usability. Instead, the appropriate repair should respect common RBAC practices and align with the context related to the access control of smart contracts.

Similar to the motivating example, RBAC is commonly implemented in smart contracts through mechanisms such as centralized role mappings (e.g., `mapping(address => bool)`), modifier-based enforcement (e.g., `onlyOwner`), and inline conditional checks using `msg.sender`. These implementations often vary significantly across contracts in structure, naming conventions, and enforcement logic. This diversity introduces challenges for automated repair, including difficulty in identifying roles due to inconsistent definitions, implicit permission logic, and the risk of introducing conflicting or redundant access checks.

C. Inspired Design of ACFIX

To address the heterogeneity of AC practices in smart contracts, ACFIX first mines common RBAC patterns from large-scale contracts and generalizes them into domain knowledge, organized as a dynamic taxonomy of role-permission pairs. The use of an external knowledge base to guide or supplement large language models is a widely recognized strategy for improving robustness and accuracy, as evidenced by recent advances in retrieval-augmented and knowledge-augmented LLMs [43], [44], [45]. Our RBAC taxonomy is designed to be continuously extensible, enabling the system to incorporate new knowledge as it encounters novel contexts. This taxonomy serves as a domain-specific external knowledge base, effectively grounding the LLM's reasoning and mitigating risks of hallucination or inconsistency. By correlating the code context of an AC-related vulnerability with the taxonomy, the LLM can infer and apply AC mechanisms that are both contextually relevant and consistent with the contract's intended logic.

Importantly, ACFIX is also RBAC-aware—it analyzes the existing enforcement pattern within the contract and adapts its

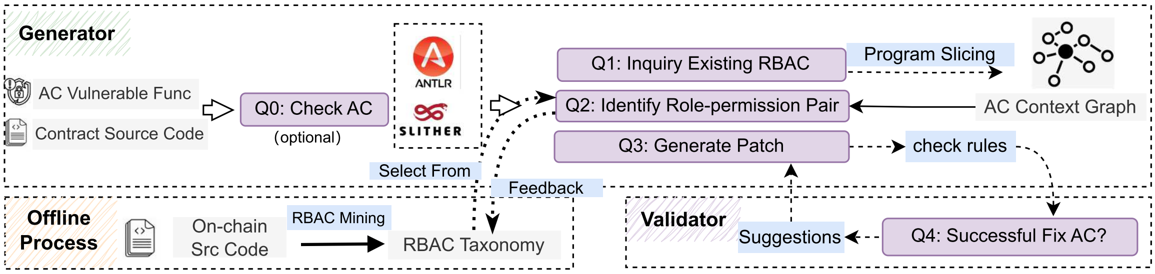


Fig. 2. A High-level overview of ACFix, consisting of both offline and online phases.

patching style accordingly. For instance, if a contract predominantly uses modifier-based enforcement, ACFix will attempt to follow this style in the generated patch to maintain semantic consistency. This adaptive behavior helps prevent structural conflicts and promotes compatibility with the original design.

Furthermore, the independent Validator Agent (*validator*) verifies whether the proposed patch aligns with the intended access control policy, ensuring that the role-permission relationship is correctly enforced based on RBAC and that the patch does not introduce functional regressions. This dual-layered approach—combining LLM-based reasoning guided by domain knowledge and semantic validation by *validator* — enables ACFix to effectively address the challenges of RBAC heterogeneity in smart contract repair. More details of this process, including the construction of the RBAC taxonomy, are further elaborated in Section IV.

III. Overview of ACFix

Fig. 2 presents a high-level overview of ACFix, which includes both offline and online phases. In the offline phase, we mine common RBAC practices from smart contracts to construct an RBAC taxonomy. This taxonomy will be used in the online phase to guide GPT-4 in pinpointing the appropriate role-permission pairs. In the online phase, for each AC vulnerability, based on the Multi-Agent Debate (MAD) architecture [29], [46], [47], [48], we employ a dual-agent architecture that consists of a *generator* and a *validator*. Specifically, we mine the RBAC taxonomy from the source code of smart contracts deployed on-chain. With this taxonomy in hand, ACFix repairs an AC vulnerability in the following steps:

1) To facilitate practicality and avoid redundant fixes, an optional step involving a checking prompt Q0 is used to confirm if the target function is subject to AC vulnerabilities. This is because while ACFix is positioned as an APR (Automatic Program Repair [49]) tool that only takes confirmed vulnerability inputs from auditing reports, CVEs, and attack incidents, we allow ACFix to be deployed as a copilot to help developers or existing AC detection tools fix potential AC vulnerabilities. In the latter case, Q0 is needed, and the result should be confirmed by an operator, as in the typical copilot scenario.
2) *Generator* then parses the contract source code, including the vulnerable part, to extract RBAC-related code elements. We then provide these elements to GPT-4 in a prompt Q1, seeking to inquire whether any element belongs to existing RBAC mechanisms in the subject code.
3) Starting from the vulnerable function f_{vul}, *generator* employs program slicing and data flow analysis to construct an inter-procedural AC Context Graph (ACG). This graph depicts the code semantically related to f_{vul}. Upon recognizing existing RBAC mechanisms in step (1), *generator* extends the ACG by incorporating relevant identifiers, such as modifiers and state variables, based on f_{vul}.
4) Using the serialized ACG as prompt Q2, *generator* guides LLMs to identify the most appropriate role-permission pair from our mined RBAC taxonomy or, if necessary, incorporates a new pair into the taxonomy.
5) After pinpointing the role-permission pair, *generator* instructs LLMs to generate a proper patch for the vulnerable code through prompt Q3. The generated patch is first statically checked for validity by rules and then continuously validated by *validator* through prompt Q4 to refine it until it is considered effective or the limit is reached.

Next, we detail the offline phase of RBAC mining in §IV and the online phase of RBAC-guided and context-aware LLM-driven repairing in §V and §VI, respectively. Regarding training or fine-tuning of LLMs, we observe that ACFix already demonstrates robust performance in repairing AC vulnerabilities by leveraging rich contextual inputs and a comprehensive taxonomy. Although fine-tuning could potentially yield incremental improvements, it introduces risks of overfitting and may reduce model flexibility. In contrast, our current design enables ACFix to dynamically incorporate newly identified RBAC pairs by updating the taxonomy, without necessitating retraining. Such adaptability cannot be achieved through fine-tuning a pre-trained model. Given these considerations, and the lack of large-scale training datasets for AC vulnerabilities, we designed ACFix to effectively combine static analysis, a pre-defined taxonomy, and in-context learning prompts to repair AC vulnerabilities without additional training or fine-tuning.

While ACFix leverages established techniques such as static slicing, code context extraction, and chain-of-thought prompting, its novelty lies in the domain-specific integration of these components to address the unique challenges of repairing AC vulnerabilities in smart contracts. Unlike general-purpose code repair, AC vulnerability repair demands precise reasoning over role-permission relationships and intricate identity checks. To address this, ACFix introduces a dynamic RBAC-guided taxonomy, a dual-agent validation-feedback mechanism, and a repair flow that directly consumes outputs from external AC detectors. This layered design enables ACFIX to not only generate patches but also validate their semantic correctness and compatibility with existing RBAC logic, offering an end-to-end, practical solution tailored for secure smart contract repair.

IV. MINING COMMON RBAC PRACTICES

During the offline phase, our goal is to systematically mine and categorize common RBAC practices observed in real-world smart contracts. Specifically, we extract role-permission pairs—the foundational elements of RBAC—from contract source code and generalize them into a structured taxonomy. This taxonomy serves as a domain-specific knowledge base that guides LLM-based repairs. By embedding this external knowledge, ACFIX can reason more effectively about access control logic and generate contextually accurate patches. Furthermore, the taxonomy is designed to be dynamically extensible at runtime, allowing ACFIX to incorporate new RBAC patterns as they emerge from usage examples.

To mine common RBAC practices, we have collected smart contracts written in Solidity [4] from $344,251$ addresses [50] on the Ethereum Mainnet as of December 2023. While we found that developers often create their own versions of RBAC, there are three major mechanisms to enforce permission checks in smart contracts:

①**OZAC**: When OpenZeppelin Access Control (OZAC) [39] is employed, roles are explicitly and uniformly implemented using templates, such as `Ownable` and `Access`. We extracted the defined roles and corresponding function names based on OZAC templates to infer permissions. ②**Modifier**: Modifier declares conditional checks that Solidity automatically embeds into the function prologues [13]. However, since modifiers can be used for various purposes, we focused only on RBAC-related modifiers that begin with `only`, such as `onlyOwner`, based on an empirical study about modifiers [13]. The roles specified after `only` and the names of *modified* functions were recognized as roles and permissions, respectively. ③**Transaction-Reverting Statements (TRS)**: The third is based on TRS [51], which use Solidity keywords, such as `require` and `if...revert`, to ensure contract integrity. A primary use of TRS is AC, where `msg.sender` is compared to predefined roles or addresses. Although TRS can serve multiple purposes, our study specifically targeted TRS assessing `msg.sender` in the context of RBAC, ensuring that our extraction remains relevant and omits distractions from unrelated uses of these statements.

Based on the three patterns above, we automatically mined $810,344$ pairs of roles and functions. After de-duplication, we identified $46,495$ unique pairs, ranked in descending order by frequency. To construct the RBAC taxonomy, we began by analyzing the top 1,000 most frequent role-permission pairs, which collectively account for 81.83% of $810,344$ all observed pairs in our data. We employed an open card-sorting methodology [52] to manually categorize permissions based on associated function names. New cards (i.e., role-permission categories) were dynamically introduced whenever a pair could not be reasonably grouped into an existing category.

The first two authors, each with over four years of experience in smart contract analysis, independently reviewed and labeled all 1,000 pairs. After individual labeling, we first merged cards that conveyed the same underlying meaning. Then, we compared the assigned cards for each pair to identify disagreements. In cases of disagreement, the final decision was made by the third author. The overall disagreement rate was 9.5%, indicating a high level of consistency between reviewers. Following this process, every one of the 1,000 pairs was assigned to a specific role-permission card, and the resulting collection formed the foundation of our final RBAC taxonomy.

Table I lists the categorized top mining results, with the first column showing the commonly used roles and the second column showing the permissions these roles may hold. We notice that these role-permission pairs are mostly related to DeFi because AC is usually implemented to manage financial assets in smart contracts. The roles could involve those with high privileges, such as *Owner of the Contract* and *Admin*, or those defined for specific operations, such as *Minter* and *Loaner*. The detailed roles depend on the usage of the contracts. It is worth noting that initially, there were 48 role-permission pairs derived from on-chain contracts in the offline process. Later during the evaluation, ACFIX dynamically updated the taxonomy and added one more pair, *Admin-Low-level Call*. The total of 49 pairs may not be exhaustive, but our evaluation showed that they have covered the majority of scenarios for which AC is implemented, and ACFIX could update it whenever new pairs are found (see Prompt Q2 in §V-C).

Based on the mined role-permission pairs, we further collected detailed permission checks for each pair from security auditing reports, as listed in the third column of Table I, which provide examples of common RBAC practices.

Revisiting the Motivating Example. With the derived taxonomy of common RBAC practices, we now revisit the motivating example in Fig. 1 to intuitively demonstrate how this taxonomy could enable ACFIX to generate the appropriate roles and permissions for real-world vulnerable code. Specifically, the function `depositFromOtherContract` could be easily matched by LLMs to the permission `Deposit` listed in Table I. Moreover, given the code context provided by our slicing in §V, LLMs can determine that this vulnerable contract has implemented two RBAC role checks, `onlyBank` and `onlyOwner`. Considering this context information and the taxonomy, LLMs could deduce the proper role-permission pair, which is `Bank-Deposit`, and generate a correct patch using the modifier `onlyBank` rather than `onlyOwner`.

V. GUIDING LLMS TO PINPOINT PROPER ROLE-PERMISSION PAIRS BASED ON CODE CONTEXT

With the common RBAC practices mined in §IV, we now use them as a "knowledge base for AC repair" to guide LLMs in fixing code with similar functionality. To help LLMs understand the functionality of subject vulnerable code that needs to be repaired, we employ static code slicing to extract AC-related

TABLE I
A TAXONOMY OF COMMON RBAC PRACTICES, FEATURING MINED ROLE-PERMISSION PAIRS AND THEIR DETAILED CHECKS

Roles	Permissions	Examples of Detailed Permission Checks
Admin	Low-level call	Multi-factor authentication
	Manage users of the contract	Multi-signature approval, Whitelisting and blacklisting, Time locks
	Manipulate price	Rate limiting, Multi-signature requirements
	Transaction management	Rate limiting, Transaction validation
	User/Role management	Regular audits, Event logging for role changes
	Utilities management	Time locks, Regular audits and testing
	Adjust fees	Validation checks for fee changes
	Monitor & analyze transactions	Access control via view functions, Data validation and sanitation
	Set trading pairs	Validation checks for trading pairs
	Configure security settings	Multi-factor authentication
Owner of the contract	Initialization	Limit initialization to authorized users against frontrun, Ensure initialization only occurs once
	Change ownership	Limit ownership change to authorized users against frontrun, Time locks
	Upgrade contract	Limit to authorized users against frontrun, Time locks, Multi-signature requirements
	Pause contract	Limit to authorized users against frontrun, Time locks
	Destroy contract	Limit destroy to authorized users, Multi-signature requirements
Owner of the funds, stakes, tokens	Burn	Validation checks for the owner of the burnable, Multi-signature control
	Claim	Validation checks for the owner of the claimable
	Withdrawal	Rate limiting, Withdrawal limits
	Swap	Transaction validation, Swap limits
	Liquidify	Rate limiting, Validation checks for liquidified funds
	Transfer	Validation checks for transferred funds
	Approve	Validation checks for privilege of approver
	Manage stakes	Validation checks for staking/unstaking
	Create pools	Validation checks for pool creation
	Set approval limits	Rate limiting
Minter	Mint	Minting limits, Whitelisting and blacklisting, Minter management, Multi-signature approval
	Setting minting parameters	Validation checks for parameters
Loaner	Offering loans	Validation checks for loan terms
	Collecting collateral	Secure handling of collateral
	Receiving payments	Transaction validation, Secure mathematical operations
	Managing defaults	Secure collateral liquidation
	Rolling loans	Validation checks for loan rollovers
	Withdrawal of funds	Limit to fund owner, Withdrawal limits, Time locks
	Viewing loan status	Data validation and sanitation
	Setting loan conditions	Validation checks for loan conditions
Borrower	Requesting loans	Validation checks for loan requests
	Depositing collateral	Secure collateral handling
	Repaying loans	Transaction validation, Secure math operations
	Managing active loans	Data validation and sanitation
	Rolling or refinancing loans	Validation checks for rollovers/refinancing
	Handling liquidations	Secure liquidation handling
	Withdrawing collateral	Validation checks for withdrawals
	Receiving notifications	Secure notification handling
Vault, Bank	Deposit	Restriction to owner of deposit, Deposit limits
	Withdrawal	Withdrawal limits, Time locks, Multi-signature approvals
	Manage funds	Rate limiting, Multi-signature approvals
	Set interest rates	Validation checks for parameters
Logger	Log	Secure storage of sensitive information Multi-signature requirements, Rate limiting
	Set log parameters	Multi-signature requirements, Using proxy patterns for upgradability and security

code context, more specifically, an AC context graph (ACG). We are particularly interested in code context related to the subject code's RBAC mechanisms. Therefore, we first leverage LLMs to identify existing RBAC mechanisms in the subject code (§V-A), enrich the code context of the identified RBAC mechanisms into ACG (§V-B), and finally instruct LLMs to use ACG to pinpoint the appropriate role-permission pair from the mined RBAC practices (§V-C). During this process, we adopt the Chain-of-Thought (CoT) [30] prompting to guide GPT-4 step by step, including the eventual AC repair generation that will be presented in the next section (§VI).

A. Identifying Existing RBAC Mechanisms

To prevent conflicts with any pre-existing RBAC mechanisms and to guide the construction of a relevant ACG in subsequent steps, ACFIX employs GPT-4 to explore existing RBAC mechanisms in the subject code, given that GPT-4 can comprehend the code. Since the names of most code elements, such as functions, state variables, and modifiers, are often self-explanatory, ACFIX extracts the names of these elements that might be associated with RBAC management. This initial information, along with the source code of vulnerable function f_{vul}, is presented to GPT-4, which is then tasked with identifying the relevant elements related to RBAC. Specifically, ACFIX first analyzes the contract to identify all pre-defined roles, permission checks, and enforcement mechanisms, including both modifier-based and inline conditional statements. If multiple AC mechanisms coexist within the same contract, ACFIX aggregates all detected enforcement styles and uses the combined RBAC structure as a reference for patch generation. When a new role-permission pair is inferred, ACFIX ensures it does not contradict or duplicate existing logic. If any overlap or redundancy is detected, the patch is generated to either update outdated logic or integrate seamlessly with the existing mechanisms in a non-redundant manner. The LLM is instructed to consider the complete set of enforcement styles to prevent the introduction of conflicting or inconsistent RBAC rules. This comprehensive analysis helps maintain a coherent and unified AC policy, even in scenarios involving multiple roles or complex, mixed enforcement implementations.

We designed our prompt based on the best practices commonly associated with using GPT-4, as suggested by [53] and [54]. Specifically, our prompt includes two parts: ① the natural language (**NL**) part that explains the task to GPT-4, and ② the code context (**CC**) part that contains the vulnerable function and other relevant code. Given that the inquiry aims to identify RBAC-related code portions, ACFIX does not include detailed code statements but only the names of relevant functions and modifiers. Following research on learning-based unit test generation [55], we include the following code context in the CC part: (1) the signature and body of the vulnerable function; (2) modifiers; (3) state variables; (4) inherited contracts; (5) functions called by the vulnerable one in sequence; and (6) any vulnerability descriptions provided in the report, if available. For the NL part, drawing upon widely recognized guidelines for using GPT-4 [56], [57], we embed: (1) a role-playing instruction (i.e., *You are a smart contract security specialist with expertise in identifying and mitigating vulnerabilities*) to inspire GPT-4's contract repairing capability; and (2) a task-description instruction to explain the task. The prompt template is illustrated in Fig. 4 for Q1.

After pinpointing specific target elements, ACFIX constructs the ACG based on them if available and f_{vul} by default.

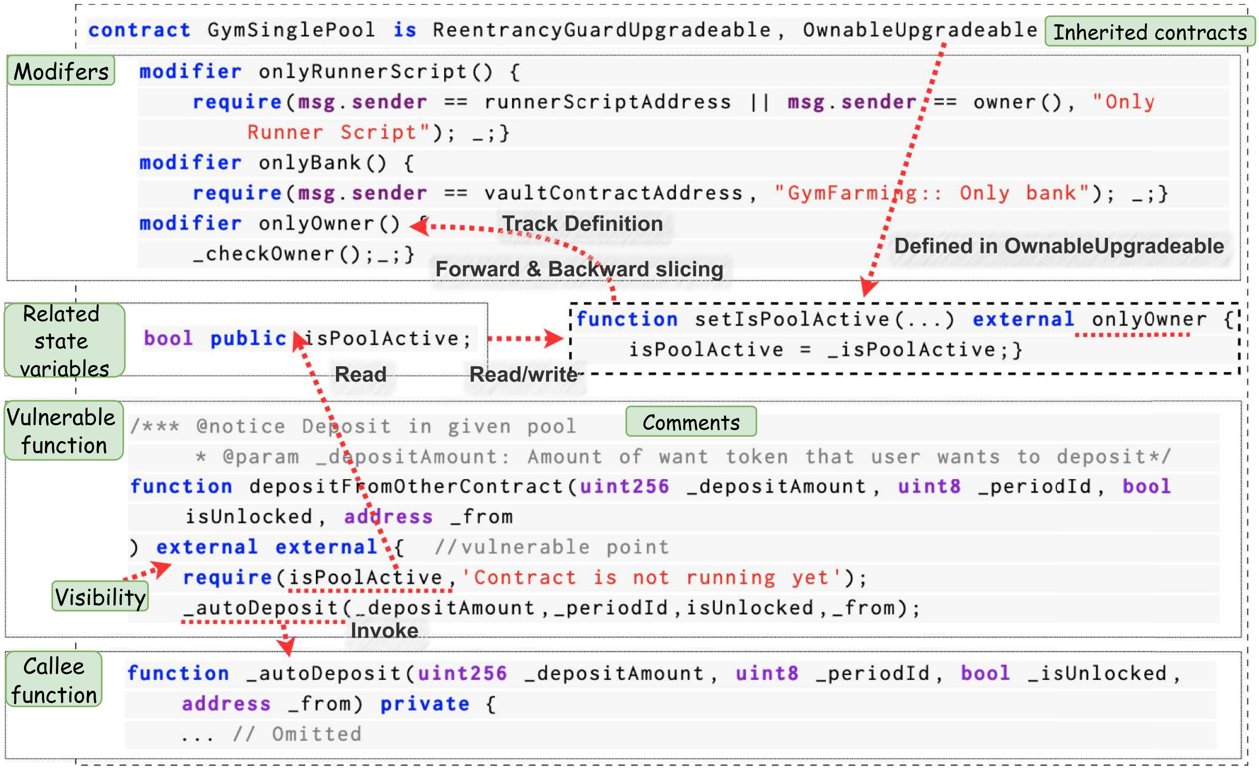


Fig. 3. AC Context graph (ACG) for the motivating example.

B. Constructing AC Context Graph (ACG)

To capture contextual code statements that constitute the functionality of the vulnerable function f_{vul}, we employ program slicing [58] as suggested by numerous previous studies [59], [60], [61], [62], [63]. Program slicing identifies code statements that influence, either through data or control, a target variable or statement. Since Ethereum-compatible blockchains [64] depend on modifications to state variables, vulnerable functions generally interact with state variables in their own or other contracts, either directly or indirectly. Based on this observation, ACFIX performs inter-procedural program slicing on the state variables interacted with by f_{vul} and associated RBAC elements (i.e., the output of §V-A). This approach aims to minimize extraneous code, ensuring a *concise* prompt that attracts focused attention from GPT-4. ACFIX, therefore, constructs an ACG that comprises a streamlined code context of f_{vul} from the subject contract.

We define ACG as $G = \{\langle V, E\rangle \,|V \subseteq \{F, Var_{state}, Mdf, Cmt\}, E \subseteq \{v_i, v_j\} | v_i, v_j \in \{f, var, mdf, cmt\}\}$, where F represents the set of functions. Var_{state} denotes the set of state variables, Mdf signifies the set of modifiers, and Cmt is the set of comments. Each vertex has three properties: *Signature*, *Body*, and the original *Contract* to which it belongs. Edges encapsulate multiple types of relationships between vertices, including *invocation*, *modifying*, *reading/writing*, and *comment*. Fig. 3 in Appendices presents an illustration of ACG for the motivating example shown in Fig. 1. Specifically, ACFIX breaks down the contract into various elements, such as modifiers and state variables, and connects them with corresponding relationships. For individual processing of elements, ACFIX performs call-chain-based inter-procedural program slicing.

To facilitate the analysis, the call graph and Program Dependency Graph (PDG) [65] are firstly constructed. Given that the input source code may not represent a complete Solidity project but rather excerpts from audit reports, it might not be compilable. Hence, program analysis tools like Slither [66] are not applicable due to their strict compilation requirements. To address this issue, we have implemented a hybrid framework that performs call graph and PDG analysis on the Abstract Syntax Tree (AST) using Antlr [67] when Slither is infeasible. Note that Intermediate Representation (IR) based analysis from Slither is preferred. Although using Antlr may result in reduced accuracy and granularity (since AST primarily captures syntactic relationships between tokens without inherent optimization, unlike the IR-based approach), it remains adequate for collecting information for this task.

However, the usage of Antlr introduces two new issues. First, unlike the three-address-code format in Slither IR, one-line source code format in Antlr might encompass multiple operators. It is necessary to split multiple operations from one statement for proper slicing. Second, it is common to accommodate the implementation within internal functions.

To address these issues, we propose several enhancements for the construction of the ACG. The general procedure of program slicing is presented in Algorithm 1. Initially, the Vulnerable Function f_{vul}, Program Dependency Graph PDG, and Call Graph CG were first calculated based on the given contract

Algorithm 1: Construction of AC Context Graph

Input: Vulnerable Function f_{vul}, Program Dependency Graph $\mathcal{PDG}$, Call Graph $\mathcal{CG}$

Output: Access Control Context Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

(1) $\mathcal{G} \leftarrow \text{Graph}(\mathcal{V}, \mathcal{E})$ // Initialize Graph
(2) $\mathcal{V}_{\text{state}} \leftarrow \text{DefUseChain}(f_{\text{vul}})$ // Extract State Variables
(3) $\mathcal{F}_{\text{callee}} \leftarrow \text{CallGraph}(f_{\text{vul}})$ // Identify Callee Functions
(4) $\mathcal{V} \leftarrow \mathcal{F}_{\text{callee}} \cup \mathcal{V}_{\text{state}}$ // Define Vertex Set
(5) **foreach** $v \in \mathcal{V}_{state}$ **do**
(6)   stmt $\leftarrow$ DefUseChain(v) // Compute Def-Use Chain
(7) $f \leftarrow$ FuncOf(stmt) // Determine Enclosing Function
(8) $\mathcal{D}_{\text{data}}, \mathcal{D}_{\text{control}} \leftarrow \{\text{stmt}\}, \{\text{stmt}\}$ // Initialize Dependencies
(9)   **while** $\mathcal{D}_{data}.next() \neq null$ **do**
(10) $stmt \leftarrow \mathcal{D}_{\text{data}}.next()$
(11)   **foreach** $opr \in stmt.split()$ **do**
(12)   **if** $v \in opr$ **then**
(13) f.AddOperation(opr)
(14)   **if** $\mathcal{PDG}.HasNextDataNode(opr)$ **then**
(15) $\mathcal{V}$.Add($\mathcal{PDG}$.NextNode(opr), f)
          $\mathcal{E}$.Add(opr, $\mathcal{PDG}$.NextNode(opr))
          $\mathcal{D}_{\text{data}}$.Add($\mathcal{PDG}$.NextNode(opr))
(16)   **else if** $\mathcal{PDG}.NextDataNode(opr) \in \{PARAMETER, RETURN\}$ **then**
(17)   callsites $\leftarrow \mathcal{CG}$.GetCallers(opr)
          $\mathcal{D}_{\text{data}}$.Add(Return(callsites))
(18)   **while** $\mathcal{D}_{control}.next() \neq null$ **do**
(19) $stmt \leftarrow \mathcal{D}_{\text{control}}.next()$
(20)   **foreach** $opr \in stmt$ **do**
(21)   **if** $v \in opr$ **then**
(22) f.AddOperation(opr)
(23)   **if** $\mathcal{PDG}.HasNextControlNode(opr)$ **then**
(24) $\mathcal{V}$.Add($\mathcal{PDG}$.NextNode(opr), f)
          $\mathcal{E}$.Add(opr, $\mathcal{PDG}$.NextNode(opr))
          $\mathcal{D}_{\text{control}}$.Add($\mathcal{PDG}$.NextNode(opr))
(25)   **else if** $\mathcal{PDG}.NextControlNode(opr) \in \{PARAMETER, RETURN\}$ **then**
(26)   callsites $\leftarrow \mathcal{CG}$.GetCallers(opr)
          $\mathcal{D}_{\text{control}}$.Add(Return(callsites))
(27) **return** $\mathcal{G}$ // Return the constructed graph

[Generator] Q1 Pattern: Inquiry Existing RBAC

- **Role playing:** You are a smart contract security specialist with expertise in identifying and mitigating vulnerabilities. **NL Part**
- **Task description:** You are provided with an issue report detailing an access control vulnerability in a Solidity contract.
- Based on the information given, analyze the vulnerability and return the code or function names that could be implemented for Role-Based Access Control.

- Vulnerable Function: *<code>*; Modifier Names: *<modifiers>*; Relevant State Variables: *<variables>*; Functions Called by Vulnerable Function: *<function names>*; Inherited Contracts: *<contract names>*; Vulnerability Description (optional): *<description>*; **CC Part**

- Pick up only the names provided above, without creating new ones. Do not explain your decision. **NL**

Fig. 4. Q1 Prompt: Existing RBAC identification.

[Generator] Q2 Pattern: Role-permission Pair Identification

- RBAC-related functions: *<signature> <sliced body> <comment>*; Callee functions are: *<signature> <sliced body> <comment>*; State variables that are read/written by the above functions: *<state variables>*; Modifiers modifies <functions>: *<name> <sliced body>* **CC Part**

- Which role and permission does the vulnerable function belong to in the following category? **NL Part**
- Always prefer the privilege that I provide. If not, name new pairs that fit to the context.
- State it clearly with format as Role: XXX, Permission: XXX. Do not explain your decision.

Fig. 5. Q2 Prompt: Role-permission pair identification.

[Generator] Q3 Pattern: Patch Generation and Validation

- The common practices of code patching for the role permission you mentioned before are *<Common practices>*.
- Your task is to provide a fix for the vulnerable function ensuring only the assigned role can execute particular function based on the common practices.
- Do not explain your decisions. Reuse existing RBAC mechanisms mentioned before if proper. **NL Part**

Fig. 6. Q3 Prompt: Patch generation and validation.

serving as the basic structure to run the algorithm for inter-procedural construction. Specifically, the initial variables are extracted and initialized in Line 1-4 and ACFix begins to iterate over the state variables Var_{state} in Line 5. For each var_{state}, the statements that read or write the var_{state} are tracked in Line 6 with the enclosing function being determined in Line 7. Next, ACFix begins slicing from statements ($stmt$) involving the state variables Var_{state} and conducts forward and backward slicing recursively by tracking dependencies related to these statements in the subsequent lines. If any operation is included in the slice, the corresponding complete line of source code is preserved in *Body*.

During slicing, ACFix recursively explores dependency chains using a Breadth-First Search (BFS) strategy, as illustrated in Lines 9–22. If a statement contains multiple operations (Line 11), it is split according to Solidity syntax using Antlr lexical patterns. Operations utilizing state variables are subsequently added to f as initial points for data flow tracing (Lines 12–13). Then, ACFix iteratively traces data and control flows, updating D_{data} and $D_{control}$ accordingly (Lines 14–15 and 23–24). For cross-function slicing, ACFix connects the parameters at function call sites with their counterparts in the function definitions, enabling backward inter-procedural slicing (Lines 16–17 and 25–26). For forward slicing, the returned variable within the function definition is linked with variable assignments receiving the function's return value at call sites. For simplicity, Algorithm 1 does not explicitly distinguish between forward and backward slicing.

C. Pinpointing the Role-Permission Pair

In this step, ACFix leverages LLMs to correlate the enriched ACG code context with common RBAC practices to identify the role-permission pair for the subject code. Due to the limited context window, ACG is serialized as the prompt for GPT-4. Specifically, elements from ACG are described in both code segments and natural language and are presented to GPT-4. ACFix first supplements the source code body for modifiers. For functions, only the statements derived from ACG are included in the body code. For state variables, the function bodies obtained from slicing are provided. Regarding inherited contracts, such as `Ownable`, the bodies of modifiers defined therein are incorporated into the prompt. In addition to these elements, edges, such as *invocation*, *modifying*, *reading/writing*, and *comment*, are all described in natural language.

Specifically, GPT-4 is prompted to select a role-permission pair from a pre-defined RBAC taxonomy. If GPT-4 identifies a pair that is not present in the current taxonomy but appears contextually appropriate, it is allowed to suggest a new pair. When such a novel pair is generated, ACFix initiates a multi-stage validation process to ensure both its relevance and uniqueness. First, the system checks for potential duplication by comparing the candidate pair with existing entries using normalized role and permission representations. This normalization process standardizes role and permission names by converting them to a consistent case, removing common prefixes or suffixes, and applying stemming or lemmatization to address minor linguistic variations. In addition, synonyms and abbreviations are mapped to unified forms using a curated dictionary and context-aware LLM prompts. By leveraging these canonicalized representations, ACFix can more accurately detect true semantic overlaps and avoid false positives. If no equivalent pair exists, the candidate is provisionally added to the taxonomy.

To further ensure the integrity and clarity of the RBAC taxonomy, ACFix periodically employs an additional language model to systematically review the entire set of pairs. This review phase is designed to identify improper, overlapping, or ambiguous entries, and to sanitize the taxonomy if necessary. When appropriate, human oversight can be incorporated to prevent unintended errors and resolve borderline cases. This consolidated review process is conducted at regular intervals, balancing cost efficiency with the need for accuracy and minimizing disruption to ongoing repair operations.

For example, during evaluation, ACFix encountered the pattern *Admin–Low-level Call*, which was not present in the original taxonomy. Recognizing its contextual relevance, ACFix successfully incorporated this pair into the taxonomy, making it available for subsequent repair tasks.

This dynamic extension, normalization, and validation mechanism enables ACFix to adapt to diverse and evolving RBAC models across smart contracts, ensuring continued relevance and extensibility.

Similar to the previous prompt, the prompt Q2 includes the CC and NL parts. The CC part is detailed with ACG information. In the NL part, a question is posed to GPT-4, asking it to select a role-permission pair from the taxonomy based on the provided code context. The prompt is as follows:

VI. Generating and Validating Patches

A. Generating Patches and Static Checking

With the appropriate role-permission pair identified in §V, ACFix now generates the final AC repair. Besides the role-permission pair stored in the LLMs' session memory from prompts Q1 and Q2, ACFix also retrieves corresponding examples of detailed permission checks from Table I to prompt GPT-4 to generate a patch. If any existing RBAC mechanisms were identified in prior responses, ACFix will prioritize reusing and enhancing them when possible to prevent any conflicts. The prompt is presented as follows:

After deriving the repaired code, ACFix conducts static grammar checks to ensure the validity of the repair. Should any discrepancies arise, ACFix consolidates these issues and relays them back to GPT-4 in a subsequent prompt, seeking an updated patch. This paper considers five kinds of static grammar checks: Avoiding Undefined Tokens, Avoiding Infeasible Function Invocations, Avoiding Misused Types, Avoiding Inconsistent Solidity Versions, and Validating the `msg.sender` Check. Details are omitted here due to page limit. Interested readers may refer to our supplementary material.

B. Generating Patches and Static Grammar Checking

After generating patches, ACFix performs a series of static and semantic checks to ensure the compatibility, correctness, and applicability of the patches before integration into the target smart contract. These checks cover both syntactic and contextual dimensions, aiming to prevent invalid or incompatible modifications that could introduce unintended behaviors. The following rules are enforced:

- **Avoiding Undefined Tokens**: ACFix first extracts all defined tokens from the current and inherited contracts, denoted as T_{defined}. Then, it analyzes the tokens introduced in the generated patch, such as new functions, modifiers, and state variables, represented as T_{repaired}. A patch passes this check only if all new tokens are properly defined or already exist.

$$\text{isDefined}(T_{\text{repaired}}) \Leftrightarrow T_{\text{repaired}} \subseteq (T_{\text{current}} \cup T_{\text{inherited}}) \quad (1)$$

- **Avoiding Infeasible Function Invocations**: GPT-generated code may call functions that do not exist or have incorrect signatures. ACFix collects the set of Solidity built-in functions $F_{\text{built-in}}$ and the user-defined functions in the repaired contract F_{repaired}, then validates that all invoked functions $Invok$ are part of this union.

$$\text{isFeasible}(Invok) \Leftrightarrow Invok \subseteq (F_{\text{built-in}} \cup F_{\text{repaired}}) \quad (2)$$

- **Avoiding Misused Types**: To prevent inconsistent or unsafe variable usage, ACFix extracts the variable types from both the original ($Type_{\text{vul}}$) and repaired ($Type_{\text{rep}}$) contracts. It ensures that variable types are used consistently and that no invalid type conversions occur.

$$\text{isConsistent}(Type_{\text{vul}}, Type_{\text{rep}}) \Leftrightarrow Type_{\text{vul}} = Type_{\text{rep}} \quad (3)$$

- **Avoiding Inconsistent Solidity Versions**: A patch may use features unavailable in the specified version of Solidity. ACFix checks whether the version required by the patch ($Version_{\text{patch}}$) is compatible with the contract's declared version ($Version_{\text{sol}}$).

$$SolCompa(Patch, SolVer) \Leftrightarrow Version_{\text{patch}} \subseteq Version_{\text{sol}} \quad (4)$$

- **Ensuring `msg.sender` Checks Are Introduced**: For access control enforcement, ACFix verifies the existence of at least one conditional statement that compares `msg.sender` to a new or existing role identifier.

$$checked(msg.sender) \Leftrightarrow \exists\, if(msg.sender == role') \tag{5}$$

- **Def-Use Chain Validation**: ACFix constructs def-use chains for all newly introduced or modified variables and ensures that each variable is correctly defined before use. This includes checking scope correctness, avoiding uninitialized variables, and ensuring no overwritten variables conflict with existing control/data flow.

$$\text{isValidDefUse}(V) \Leftrightarrow V_{\text{patch}}^{\text{use}} \subseteq (V_{\text{defined}} \cup V_{\text{patch}}^{\text{def}}) \tag{6}$$

- **Structural Compatibility Check**: Before applying the patch, ACFix validates that the modified code block aligns with the structural boundaries of the original smart contract. For example, a function-level patch must respect existing function signatures and modifiers.

$$\begin{aligned} \text{Compatible}(S_{\text{patch}}, S_{\text{target}}) \Leftrightarrow \forall s \in S_{\text{patch}}, \\ \text{ValidContext}(s, S_{\text{target}}) \end{aligned} \tag{7}$$

This multi-step validation pipeline enables ACFix to generate patches that are not only grammatically valid but also semantically consistent and directly applicable to the original smart contract codebase.

C. Validating Patches' Effectiveness via MAD

Once all static and rule-based checks are passed, ACFix engages the Validator Agent (*validator*) to perform a higher-level semantic validation of the patch's effectiveness through a multi-agent debating (MAD) loop. This step is essential to ensure not only syntactic correctness but also functional and security alignment with the intended AC policy. In this process, the Generator Agent (*generator*) first outputs a candidate patch and provides it to *validator* along with the vulnerability description, the surrounding code context, and the selected role-permission pair. The Validator Agent independently evaluates whether the patch (1) correctly mitigates the identified AC vulnerability, (2) preserves the original contract logic, and (3) does not introduce any new security or logical flaws.

The *validator* performs this assessment by simulating the review process a domain expert might conduct. It reasons over the vulnerability description and the repaired code to determine if the AC logic is properly enforced—e.g., checking that access is restricted to intended roles, permission boundaries are respected, and the role-permission pair selected by *generator* is consistent with the contextual semantics. If the patch is deemed insufficient or flawed, *validator* returns structured feedback, including the reason for rejection (e.g., incorrect role, missing validation, logic conflict). This feedback is then passed to *generator*, which uses the information to refine and regenerate an improved patch. This repair-validation cycle continues in a loop with a maximum of 3 iterations to balance thoroughness and efficiency.

Even if the patch is not accepted after 3 attempts, the last generated patch is retained as the final output. Based on our empirical evaluation (see Section VII-C), this iterative mechanism proves highly effective: over 90.9% of the cases required at most one re-attempt, and only a single case failed to pass validation after three rounds. This agent-based validation framework strengthens ACFix by introducing a self-regulating feedback loop that improves patch robustness, reduces hallucinations, and ensures more consistent adherence to access control principles.

VII. Evaluation

We aim to evaluate ACFix based on its effectiveness in appropriately repairing AC vulnerabilities by answering the following six research questions (RQs):

RQ1: LLM Selection. *How do popular LLMs perform as the base model of* ACFix *and which is the best?*
Given the emergence of multiple LLMs offering similar code generation capabilities, we first needed to evaluate these models to determine the most suitable base model for ACFix. To achieve this, we assessed all popular LLMs available as of the submission date, comparing their performance as query interaction models within ACFix using a benchmark dataset. Specifically, we focused on two primary metrics: generation rate and success rate.

RQ2: Effectiveness Analysis. *How effectively does* ACFix *repair AC vulnerabilities compare to other vulnerability repairing tools for smart contracts?*
After selecting the best-performing model, we conducted a comprehensive and fair comparison with existing smart contract repair tools. To facilitate this, we first created the initial benchmark dataset specifically tailored for AC vulnerabilities. Using this dataset, we evaluated all available tools, identifying their strengths and weaknesses. Furthermore, we analyzed failure cases to understand and reveal the underlying reasons behind incorrect repairs.

RQ3: Ablation Analysis. *How does the performance of ACFix compared to a baseline that uses only GPT-4 with raw code and descriptions as input?*
To evaluate the contribution of each procedure implemented in ACFix, we systematically masked individual components to clearly highlight their respective impacts. Additionally, we compared ACFix against the vanilla GPT-4 model to emphasize the effectiveness and the novel design beyond the capabilities of the base model.

RQ4: Effectiveness by Categories. *How do tools perform across various categories in the benchmark dataset?*
The complexity of repairing AC vulnerabilities largely depends on accurately discerning nuanced differences between candidate roles and comprehending the contextual meaning within the code. Consequently, the difficulty of cases within our benchmark dataset varies significantly. To gain deeper insights into the performance of ACFix across varying levels of difficulty, we further categorized and analyzed these cases based on their ground truth RBAC pairs.

TABLE II
SOURCES OF THE BENCHMARK DATASET

Source	NVD	DefiHackLabs	tintinweb	SmartFix	Media
Count	19 ([69])	28 ([70])	60 ([71])	8 ([14])	3

RQ5: Practicality Analysis. *Is* ACFIX *able to check potential vulnerabilities reported by static checkers?*
Since ACFIX is designed to repair vulnerabilities reported by vulnerability detectors, we conducted this experiment specifically to evaluate such practical scenarios. To demonstrate its effectiveness, we integrated ACFIX with three widely-used static analysis tools capable of detecting AC vulnerabilities, thereby simulating an end-to-end workflow for vulnerability handling. This step evaluates the Q0 component of ACFIX, ensuring its practical capability to correctly process inputs provided by vulnerability detectors.
RQ6: Efficiency Analysis. *How does* ACFIX *perform in terms of efficiency and financial cost?*
Considering both execution time and monetary cost, an LLM-based tool such as ACFIX is expected to deliver efficient and cost-effective vulnerability repairs with notable results. Therefore, we conducted an end-to-end evaluation, systematically monitoring the execution time and monetary expenses associated with using ACFIX.
Data Preparation. We constructed our dataset by building upon existing research, starting with 19 Common Vulnerability Enumerations (CVEs) that are frequently cited in prior AC-related studies [9], [12], [13]. For reference, SPCon and SoMo [12], [13] each evaluated on 44 cases, AChecker [9] used 21 cases, and SmartFix [14] focused on just 12 AC-related cases. While the SmartBugs dataset [68] is commonly used in broader smart contract vulnerability research, it was excluded from our evaluation due to the absence of ground truth annotations identifying whether the cases involve access control vulnerabilities. As our focus is specifically on AC vulnerabilities, such omissions make SmartBugs unsuitable for inclusion.

However, relying solely on CVEs does not yield a comprehensive evaluation. Given the absence of a benchmark dataset for AC vulnerabilities, we introduce the first benchmark dataset of real-world instances with ground truths. This dataset has been assembled from five primary sources as indicated in Table II (already covering the sources from the above-mentioned work): ① 19 CVEs from NVD [69]. ② Defi Hack Labs [70] has published numerous vulnerabilities with real-world attacks. Under the "Access Control" category, we collected 28 cases with vulnerable code snippets and blockchain addresses. ③ An open vulnerability dataset provided by tintinweb [71] contains 28,699 vulnerabilities sourced from real-world auditing reports. After filtering for "Access Control," we identified 60 unique cases. ④ The dataset from SmartFix [14] includes 8 AC cases related to the misuse of `tx.origin`. ⑤ Additionally, we collected 3 more cases from media sources, including BlockSec [72], SlowMist [73], and Medium [74]. In total, we have compiled 118 real-world cases, making it the most extensive publicly available AC vulnerability dataset to date [75].

Metrics. Given that evaluating the correctness of patches remains a challenge in Automatic Program Repair [49], determining whether a repair is appropriate for the contract without overfitting involves leveraging multiple metrics to evaluate repairers. The following metrics were used for evaluation:

- **Comparison with Author Fixes**: Due to security concerns, many DeFi organizations and teams refrain from publishing the corrected code post-attack. We managed to collect 20 real fixes by the original authors to serve as target repairs for these 20 cases. Any patch that diverged from these original fixes was deemed unsuccessful.
- **Exploitation-Based Evaluation**: DeFi Hack Labs [70] provides exploitation scripts that demonstrate how vulnerabilities can be exploited in a simulated environment, using authentic contracts sourced from the blockchain. We used these scripts to determine whether the vulnerability remains exploitable after the repair. We ran exploit scripts on both the original and repaired code to demonstrate that the repaired contracts are no longer exploitable. The logs for both of them are provided in our dataset [75].
- **Manual Inspection**: The first two authors manually examined the repaired contracts to determine if the patch was appropriate. The third author made the final decision in the event of a disagreement. The explanatory notes are listed in our dataset [75].

It is worth noting that our initial intention was to utilize detection tools to determine whether the AC vulnerability still existed after repairs. However, no suitable tool was found to work properly for the cases within our dataset (except for 19 CVEs). Specifically, AChecker [9] works only for bytecode contracts. When we ran AChecker against 43 compilable AC cases, only 3 were detected (with testing logs recorded on our website [75]), leading to its exclusion from the evaluation. SPCon [12] requires transaction history, and SoMo [13] targets only modifier-based AC vulnerabilities and has yet to release its source code. As for other generic detectors such as Securify [76] and Slither [66], they require either compilable source code or precompiled bytecode, with the exception of SmartCheck [77]. However, upon running SmartCheck on our dataset, we found that it generated many false alarms about other types of vulnerabilities but very few concerning AC, indicating its unsuitability for detecting AC vulnerabilities.
SOTA Repair Tools to Be Evaluated. Various repair tools for smart contracts have been proposed in recent years. We selected benchmark tools through a principled selection process. Initially, we searched for papers using keywords such as "smart contract" and "security" in top-tier security/software engineering/programming language venues from the past three years (up to June 2024), yielding 268 papers on smart contract security. Excluding papers unrelated to vulnerability fixing, 15 relevant papers were derived.

From these papers, we identified 9 baseline candidates, including SGuard [15], SGuard+ [78], SmartShield [16], SCRepair [17], Elysium [19], Aroc [18], HCC [79], EVMPatch [20], SmartFix (2023) [14], ContractTinker [80], and LLMSmartSec [81]. We then excluded three tools that were either inapplicable for our patch generation or unavailable, and four tools that only work on bytecode, resulting in a final list of three repair tools for source code. Specifically, the artifact for HCC is not available. Since SmartShield, Aroc, and Elysium are designed exclusively for bytecode repair, they were omitted from our comparative study. Meanwhile, SCRepair requires manually curated unit tests for patch generation, a resource that our dataset lacks. Among these tools, only SGuard and SmartFix have available artifacts and are capable of accepting source code and repairing AC vulnerabilities, leading to their inclusion in our analysis. ContractTinker is an LLM-based smart contract repair tool designed to handle various vulnerability types without being limited to a specific category. We included it in our evaluation by adapting its input pipeline to process vulnerability descriptions in our dataset. The modified ContractTinker is denoted as ContractTinker∗. LLMSmartSec is another recent LLM-based approach aimed at secure smart contract generation and repair. However, at the time of our evaluation, it lacked runnable artifacts and clear documentation, making reproduction infeasible. SGuard+ extends the rule-based repair engine of SGuard with enhanced capabilities. However, it does not provide public implementation or configuration files, making it impractical to replicate its repair logic without introducing bias. Therefore, among these, SmartFix, SGuard, and ContractTinker were included as baselines in our evaluation to ensure a fair and reproducible comparison.

A. RQ1: Pilot Study to Identify Suitable LLM

Given the various available LLMs, we first tested several popular and state-of-the-art models, including GPT-4 [33], GPT-3.5 [34], Mistral [35], and LLaMA 3 [36], to select the base LLM for ACFix. In our updated evaluation, we further integrated LLaMA 3.2–11B, the latest lightweight variant of the LLaMA family, to enable a fair and up-to-date comparison with GPT-4. All LLMs were implemented under the same evaluation pipeline, with the only differences being in output formatting. The OpenAI API allows for structured response formatting [82], making output parsing straightforward for GPT-4 and GPT-3.5. In contrast, although we explicitly instructed Mistral and LLaMA models to respond in JSON format, consistent compliance could not be guaranteed. Therefore, we implemented a robust string-based parser to reliably extract structured outputs across all models. We chose not to include GPT-4o in this comparison due to potential concerns around training-time data leakage and limited control over evaluation consistency. Including such models may lead to non-reproducible or unfair results, particularly in security-sensitive tasks like access control repair. The comparative results between GPT-4 and LLaMA 3.2–11B are highlighted in Table III, showcasing their respective performance in terms of patch accuracy, runtime, and model responsiveness.

To avoid data leakage, we selected the LLMs with the earliest cutoff dates. For GPT-4, the model was `GPT-4-0613` (training data up to September 2021). GPT-3.5 was `GPT-3.5-turbo` (also up to September 2021). As Mistral and Llama3 were released more recently, the earliest models that we could find

TABLE III
REPAIR RESULTS FOR THE POPULAR BASE LLMS

Model	$\#Generated$	$\#Success$	$Rate_{gen}$	$Rate_{success}$
GPT-4	**118**	**112**	**100.00%**	**94.92%**
GPT-3.5	115	66	97.46%	55.93%
Mistral-7b	113	58	95.76%	49.15%
Llama3-8b	117	87	99.15%	73.72%

$\#Generated$ is the number of cases in which patches were generated. $\#Success$ is the number of cases that a correct patch is successfully generated passing 3 metrics.

were from October 2023 and May 2024, respectively. Therefore, these two models were trained with newer data, potentially leading to data leakage and enhancing their capabilities in evaluation. The configurations for these LLMs were all set to a temperature of 0 (to suppress randomness) and a maximum of 4096 tokens for output.

Table III presents the results of comparison among LLMs. The GPT-4 model has demonstrated an excellent ability to provide correct patches for AC vulnerabilities, as evidenced by its much higher $Rate_{success}$. This proficiency stems from its reasoning ability to deduce the proper role-permission pairs. After manually reviewing the failed cases of other LLMs, most were found to be caused by over-fitting roles, such as *the owner of the contract*. The difference in selected role-permission pairs among LLMs has exhibited their varying abilities to summarize roles by understanding the source code context. Another major category of failed cases resulted from grammar mistakes leading to uncompilability. Cases where patches were not generated were caused by requests rejected by the LLMs to generate patches. Based on the overall results and the above analysis, other models except GPT-4 exhibit sub-optimal context comprehending, unreliable generated code, and rejected requests. Thus we selected GPT-4 as the base model for ACFix, as mentioned earlier in §II.

> **Answer to RQ1**: Given its superior performance in generating appropriate patches for AC vulnerabilities compared to three other popular LLMs, GPT-4 was chosen as the base model for ACFix.

B. RQ2: Evaluating ACFix and SOTA Tools

ACFix, SmartFix, and SGuard were run on our benchmark dataset to generate patches. We first checked the compilability of the patches. Then, we evaluated the correctness of the patches using three metrics. We introduced the term *Generate Rate* ($Rate_{gen}$) to denote the percentage of generated patches across all cases, and *Success Rate* ($Rate_{success}$) to represent the proportion of patches that meet the three criteria of the stipulated metrics as successful repairs. As illustrated in Table IV, ACFix was able to generate patches for all 118 cases, with 112 of them considered successful repairs, resulting in a *Success Rate* of 94.92%. In contrast, SGuard could only generate patches for 6 case, and SmartFix for 21 cases. The analysis of results and

TABLE IV
REPAIR RESULTS OF TOOLS IN THE BENCHMARK DATASET

Tool	*#Generated*	*#Success*	$Rate_{gen}$	$Rate_{success}$
ACFix	**118**	**112**	**100.00%**	**94.92%**
SGuard	6	1	5.08%	0.85%
SmartFix	21	7	17.80%	5.93%
ContractTinker*	6	0	5.08%	0.00%
W/o ACG	113	106	95.76%	89.83%
W/o RBAC	118	81	100.00%	68.64%
W/o Validator	118	103	100.00%	87.28%
Vanilla GPT-4	113	62	95.76%	52.54%

reasons behind the performance of all tools will be elaborated upon.

Analysis of Results from ACFIX. Out of the 112 successfully repaired cases, their compilability was checked against 43 cases that were already compilable before patching. It turned out that all of them could be successfully compiled with the corresponding Solidity versions. As for the 6 unsuccessful repair cases, we categorized them into four reasons: (4 cases) **Over-protection** (overfitting): ACFIX returned repairs that could potentially hinder the routine usage of certain users. For example, ACFIX repaired a contract that allowed anyone to steal the collateral of loaners by adding an `onlyOwner` modifier, which restricted access from normal loaners who were supposed to be authorized to claim their own collaterals. One case was caused by insufficient context provided from the context extraction step, so GPT-4 could not recognize the correct permissions. The other three were caused by the strict *validator* that prefers conservative measures. (1 case) **Different from Real Fixes**: For most cases with real fixes, ACFIX performed well by providing the same protection as the real fixes. However, there was a case where the real fixes considered non-code information, which ACFIX could not predict from merely a code-based context. For example, the function `safeTransferFrom` was changed to `internal` from `external` after fixing, without any clear reason provided in the code. This change could potentially overfit against legitimate users. (1 case) **Unclear Requirements of the Description**: The description of this vulnerability indicated only insufficient checks for potential users. Indeed, it required multiple checks for the arguments in addition to the `msg.sender` to ensure proper functionality. ACFIX failed to provide sufficient checks for this vulnerability.

Analysis of Results from SOTA Tools. As illustrated in Table IV, SGuard [15] could only generate fixes for 6 cases, and 1 of them passed the three metrics. SmartFix [14] managed to generate 21 fixes with 7 successful ones. The primary reason for the failed cases of both tools is compilation failure because they depend on IR derived from compiled code. However, sources for some AC vulnerability cases have not released on-chain addresses but only vulnerable code snippets. Even when addresses are provided, the source code may not be disclosed by blockchain explorers such as Etherscan [83]. The analysis of the tools is elaborated as follows:

SGuard: All cases that were not generated were due to unsuccessful compilation, as logged by SGuard. Out of the 6 patches generated by SGuard, 5 failed to repair the AC vulnerability.

[Validator] Validate Patches

- **First round:** Can this patch fix the vulnerability? <patch from generator>
- The source code is <source code>. State the answer and reasons.

NL Part

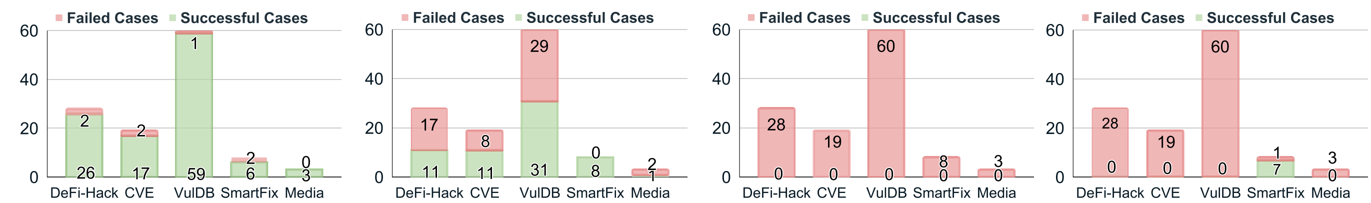
- **Second round onwards:** The patch is updated as <new patch>.

Fig. 7. Q4 Prompt: Patch validation.

Four of these failed cases had patches that were exactly the same as the vulnerable code, indicating that SGuard failed to identify the necessary fixes. For the remaining case, SGuard provided a fix that was irrelevant to AC. The only case correctly repaired involved the misuse of `tx.origin`, suggesting that SGuard was specifically designed to address `tx.origin` misuse in the context of AC vulnerabilities.

SmartFix: SmartFix generated patches for 21 cases, accounting for 17.80% of AC vulnerabilities. However, only 7 of them successfully fixed AC vulnerabilities, all of which were cases of misuse of `tx.origin`. Among the unsuccessful repairs, none of the 14 cases were related to `tx.origin` but to other types, as illustrated in Fig. 8(d). Out of 14 unsuccessful patches, 13 targeted other non-AC vulnerabilities, including 12 cases of Integer Over/underflow and 1 case of Reentrancy, but left AC vulnerabilities unrepaired, which did not exist in the original contracts upon manually examination. SmartFix only accurately identified the AC vulnerabilities in two cases, both related to re-initialization issues. In these cases, SmartFix replaced the incorrectly named constructor function with the Solidity keyword `constructor`, without considering that the `pragma` versions were both `^4.x.x`, which does not support the `constructor` keyword. As this fix would lead to compilation failure, we labeled them as unsuccessful fixes. The overall result shows that SmartFix was designed to repair AC vulnerabilities, but its effectiveness is limited to types of AC such as re-initialization and misuse of `tx.origin`.

ContractTinker*: Among the 118 cases in our dataset, only 43 contracts were compilable and thus eligible for processing due to ContractTinker*'s reliance on Slither for code analysis. Moreover, the tool expects structured vulnerability reports (e.g., 'HighRiskFindings'), which were not consistently available in our dataset. After modifying its input pipeline to accept natural-language vulnerability descriptions, ContractTinker* generated patches for only 6 functions across 8 output files, accounting for just 5.08% of the total dataset. Manual inspection of the outputs revealed that none of the generated patches correctly repaired the access control vulnerabilities. Only one fix is related to AC but produces an overfitting change, i.e., replacing 'public' with 'private'. The rest of the fixes were unrelated, including zero-address checks and balance checks. Therefore, the effective success rate of ContractTinker∗ in repairing AC vulnerabilities in our benchmark was 0%. We attribute this low effectiveness to ContractTinker*'s design, which is built around a direct conversation with the LLM, without external knowledge integration or domain-specific context. As a result, its performance heavily depends on the availability of clean, structured vulnerability reports. However, in practice, such structured reports are often inconsistent or missing entirely, limiting the tool's applicability in real-world repair scenarios.



(a) ACFix (b) *Vanilla GPT-4* (c) SGuard (d) SmartFix

Fig. 8. Effectiveness of tools on various data sources.

Baseline Tool Using Vanilla GPT-4

Role playing and task description: NL Part
- You are a smart contract security specialist with expertise in identifying and mitigating vulnerabilities.
- You are provided with an issue report detailing an access control vulnerability in a Solidity contract.
- Your task is to provide a fix for the vulnerable function ensuring only the proper role can execute particular functions. Do not explaining the rationale behind your decisions.

- Vulnerable Code: *<code>* CC Part
- Vulnerability Description: *<description>*

Fig. 9. Baseline tool using vanilla GPT-4.

> **Answer to RQ2**: ACFix successfully generated repairs for 100% of AC vulnerabilities, effectively fixing 112 cases, representing a 94.92% success rate. This demonstrates that ACFix can repair the majority of AC vulnerabilities across a variety of scenarios. It also outperforms SOTA contract repair tools, SGuard and SmartFix, which only successfully repaired the misuse of `tx.origin` and could not handle AC vulnerabilities in broader scenarios.

C. RQ3: Ablation Study

To demonstrate the effectiveness of the RBAC taxonomy, context information, and the MAD mechanism, we conducted an ablation study based on four customized baselines. We iteratively removed individual components of ACFix. Specifically, *W/o ACG* has the same implementation as ACFix but without ACG. *W/o RBAC* lacks the RBAC taxonomy. *W/o Validator* solely utilizes *generator* without *validator*. *Vanilla GPT-4* uses `GPT-4-0613` with raw vulnerable code and vulnerability descriptions directly, without preprocessing, as in Fig. 9.

As shown in Table IV, *W/o ACG* and *Vanilla GPT-4* generated patches for 113 cases instead of 118 because five contracts have multiple source code files that exceeded the token limit. Furthermore, *W/o ACG* fixed 106 cases, indicating that ACG contributed to 12 more successful cases. The number is not significant as most of the contracts were retrieved from auditing reports, which have limited length. However, ACFix could benefit more from ACG in terms of scalability for real-world deployed contracts. The 81 successfully fixed cases demonstrate that RBAC taxonomy has significantly contributed to the patch generation. The taxonomy can be dynamically updated when new pairs are encountered by ACFix, such that *Admin-Low-level Call* was added by ACFix during evaluation. It has been substantiated that GPT performs better for patch generation if the generation is guided by well-structured knowledge.

> **Answer to RQ3 for *W/o ACG* and *W/o RBAC***: The comparison with these two baselines have substantiated that ACG and the RBAC taxonomy could improve the repair by fixing an additional 12 and 37 cases, respectively.

For *W/o Validator*, without *validator*, 15 patches were not generated correctly. It was observed that *validator* successfully validated 9 more patches, resulting in correct patches. The errors in 5 of these patches were previously due to misalignment with the vulnerability description, while another 4 were due to overfitting roles. Fortunately, they were corrected after review by *validator*, meaning that MAD can effectively correct improper patches through independent evaluation.

Regarding the number of MAD loops,Within the 118 cases, ACFix completed the generation after 0, 1, 2, and 3 re-attempts for 41, 68, 7, and 2 cases, respectively. 92.37% of cases were completed within 1 attempt. This demonstrates that MAD typically converges quickly within 3 loops.

However, *validator* was observed to introduce over-fitting patches in some instances, in addition to correcting others. ACFix failed in 3 cases due to over-fitting checks. After scrutinizing the history of debates between agents, it was found that the patches were initially correct as generated by *generator*. However, *generator* was persuaded to adopt conservative roles like `owner` by *validator* after debate. Therefore, even with *validator*, determining the appropriate role-permission pair is stillchallenging. Still, *validator* could effectively safeguard the output according to the evaluation.

> **Answer to RQ3 for *W/o Validator***: *W/o Validator* failed to fix 9 cases compared to ACFix, suggesting that $Rate_{success}$ could be further boosted with *validator*.

Vanilla GPT-4 has successfully repaired 62 cases (52.54%). We manually analyzed the distribution of the repaired cases and found that *Vanilla GPT-4* tends to apply conservative roles in the repairs (68.64% of the total), such as `onlyOwner`. For 40 out of the 60 successful cases, *Vanilla GPT-4* generated repairs using `onlyOwner`. In another 17 cases, the ideal roles were specified in the vulnerability descriptions, allowing *Vanilla GPT-4* to directly reuse the given roles. For the remaining 3 cases, the function signatures themselves provided enough context for GPT-4 to infer potential roles, such as *borrower* from the function `borrow`. In contrast, out of the 58 incorrect repairs, 37 were inaccurately over-protected by `onlyOwner`, affecting legitimate users. The rest of the cases were deemed improper because they were either still vulnerable or uncompilable.

> **Answer to RQ3 for *Vanilla GPT-4***: Without the RBAC taxonomy and ACG, *Vanilla GPT-4* achieved a repair success rate of only 52.54%. This highlights the vital importance of the ACG mined by ACFIX from the code and the guidance provided by the RBAC taxonomy.

Besides the two baselines, we further explored the effectiveness of *generator* rule checks. Patches of 4 cases violated the rules in §VI-A. Upon manual inspection, it was determined that 2 cases involved incompatible pragma versions, and the other 2 were related to mis-spelled variable names, which could be attributed to LLM hallucination or loss of focus [27]. However, they did not affect the effectiveness of ACFIX, considering that the rule checks could safeguard the output.

D. RQ4: Effectiveness by Categories of Roles

The complexity of repairing AC vulnerabilities depends largely on accurately distinguishing nuanced role differences and interpreting the code context, resulting in varied difficulty levels across our benchmark dataset. To better understand ACFIX's performance under different complexity scenarios, we categorized and analyzed benchmark cases based on the roles. After manually annotating the appropriate role-permission pairs for each vulnerability in the benchmark dataset, we further categorized the 118 AC vulnerabilities according to their corresponding roles.

In this evaluation, the ground truths of our benchmark dataset were mapped to 31 out of 49 entries of the taxonomy, demonstrating that our taxonomy captures a wide range of AC patterns and is not overfitted to a narrow scope. This further confirms the dataset's diversity and the generality of the taxonomy itself. As permissions may vary from case to case, we focused Fig. 10 solely on eight major roles regarding the proportion of successful repairs. It was observed that three major roles—*Owner of the Contract* (OC), *Owner of Funds* (OF), and *Admin*—account for the majority (77.12%) of the AC vulnerability benchmark dataset. Generally, ACFIX achieved the best repairs across the eight roles, but its performance for the roles of *OC* and *Admin* was less effective. These roles usually have the broadest range of permissions, and *validator* tends to encourage *generator* to adopt conservative roles, such as *OC* and *Admin*. This is evidenced by *W/o Validator*, which achieved slightly better results for the role of *OC* (98% v.s. 94%). Since *Vanilla GPT-4* lacks refined context, it performs worse than ACFIX across all roles.

> **Answer to RQ4**: ACFIX struggled with the roles of *OC* and *Admin* but still outperformed *Vanilla GPT-4* across all roles. On the other hand, SmartFix was only able to repair 17% of the *Initialization* cases.

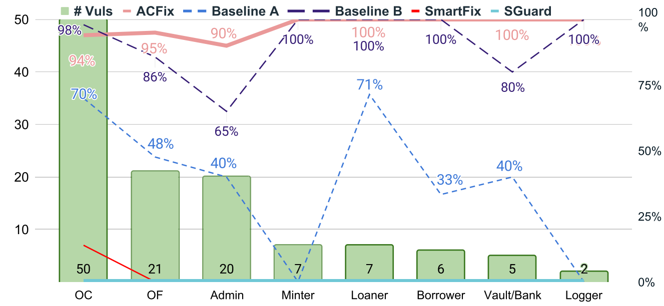


Fig. 10. Proportion of successful repairs by roles.

E. RQ5: Practicality Analysis

As described in §III, ACFIX is designed to function as a copilot for processing vulnerability reports generated by external AC vulnerability detectors. It is not intended to serve as a standalone detector, instead, it operates downstream by consuming flagged outputs, such as potentially vulnerable functions, and assisting in precise vulnerability localization and automated patch generation. Given the high computational cost of large-scale contract analysis, ACFIX is deliberately scoped to handle a limited set of contracts identified by existing detection tools. To evaluate its effectiveness in this role, particularly its ability to accurately confirm AC vulnerabilities and reduce human verification effort, we assessed the Q0 step of ACFIX on cases reported by three state-of-the-art AC vulnerability detectors.

We selected three tools that publicly provide labeled TP and FP cases: Slither [66], SoMo [13], and SPCon [12]. AChecker [9] was excluded due to the lack of a publicly available dataset. For true positives, we carefully curated 21 TP cases confirmed by the original tools in their released datasets. Since these tools independently verified these vulnerabilities, they serve as a reliable basis for TP evaluation.

To simulate integration with vulnerability scanners, we provided ACFIX with only the vulnerable functions (as detected by the tools) and no further textual description. ACFIX was then tasked with identifying the vulnerable lines and determining whether the root cause was a missing or incorrect identity check (e.g., `msg.sender`). Successful identification under this setting validates ACFIX 's capability to localize and explain the vulnerability, making it a practical companion to detection tools.

Regarding false positives, due to the scarcity of public FP cases from SoMo, we collected 10 cases from its dataset. To maintain balance, we randomly selected 10 FP cases each from Slither and SPCon, resulting in a total of 30 FP cases. Combined with the 21 TPs, the full evaluation consists of 51 unique cases. This expanded and clarified evaluation setup strengthens the reliability and interpretability of RQ5.

As shown in Table V, the Q0 step of ACFIX has been evaluated on these 40 positive cases. If Q0 is able to accurately determine the actual vulnerability status of each case, the case is considered correctly identified by ACFIX. Specifically, the case is correct if Q0 returns True for a TP case

TABLE V
THE Q0 ANALYSIS RESULTS FOR 40 POSITIVE (TP/FP) CASES REPORTED BY THREE SOTA AC VULNERABILITY DETECTORS

Detectors	**All TP**	**Slither FP**	**SoMo FP**	**SPCon FP**
Correct/Total	21/21	7/10	10/10	9/10

TABLE VI
MONETARY AND TEMPORAL COSTS OF ACFIX

Name	**Avg. Token**	**Avg. Total Price (USD)**	**Avg. Total Time (s)**
ACFix	1,956.35	0.0588/6.9429	30.58/3,608.23
W/o ACG	2,813.68	0.0843/9.9474	37.23/4,393.16
W/o RBAC	1,845.84	0.0552/6.5341	29.35/3,463.31
W/o Validator	1,777.90	0.0546/6.4428	25.23/2,977.14
Vanilla GPT-4	378.79	0.0192/2.2542	7.66/903.98

and False for an FP case. It is shown that ACFix could correctly identify most cases (36/40). Although 4 FP cases were not correctly identified by Q0, no TP cases were missed by ACFix, thus ensuring that real AC vulnerabilities could be fixed.

> **Answer to RQ5**: Out of 40 positive AC vulnerabilities reported by SOTA tools, ACFix correctly identified 36 of them, demonstrating its practical value in confirming and fixing AC vulnerabilities.

F. RQ6: Cost Efficiency and Performance

Table VI shows the monetary and time costs of using ACFix for all AC cases in the dataset. Regarding monetary cost, the average number of tokens used across two agents by ACFix is $1,956.35$. According to the current pricing plan [84], the average cost for repairing one AC vulnerability is 0.0587 USD. Consequently, repairing all vulnerabilities in the dataset costs a total of 6.9266 USD. Note that the token counts for ACFix and *W/o Validator* were much higher than for *Vanilla GPT-4* because the costs of failed cases were not counted, and consecutive conversations require incorporating the previous history into the new prompt, which results in repeated counting of tokens. Additionally, it is evident that *W/o ACG* consumed more tokens than ACFix because the raw source code was not processed to highlight critical information by constructing ACG. Instead, the raw source code of contracts was incorporated in the prompt, including redundant tokens, leading to unnecessary costs and inefficiency. Regarding temporal cost, the average time for ACFix to patch each AC case is 30.58 seconds. The time required for static analysis may vary depending on each case's complexity.

> **Answer to RQ6**: On average, ACFix costs 0.06 USD and takes 30.58 seconds per case.

VIII. LLM-BASED REPAIR VS. HUMAN REPAIR

Following the evaluation of ACFix itself in §VII, we further proceed to understand the value of ACFix's repairs from a human perspective, e.g., how they align with human repairs and whether they are non-trivial to devise by humans (if non-trivial, this means that ACFix provides a unique complement to assist human-in-the-loop repair as a copilot).

Towards this objective, we conducted a human-based evaluation involving 10 practioners who have worked on smart contract auditing for 2-7 years. They are divided into junior (2-4 years) and senior (4-7 years) groups. Given the raw source code and vulnerability description, the participants were asked two questions: ① Write down the most appropriate role-permission pairs they thought fit the situation; ② Indicate if the patch is straightforward to come up with based on their understanding. For ②, unless explicitly stated, they were not asked to produce patches but only to assess the difficulty, because manually curated patches are hard to normalize for comparison. Note that as this study does not involve any personally identifiable information, the IRB (Institutional Review Board) requirement was waived by our institution.

A. How ACFix's Repairs Align with Humans

After manually scrutinizing the role-permission pairs curated by experts, 83 (74%) and 69 (62%) pairs by senior and junior experts respectively were aligned with pairs produced by ACFix for 112 corrected cases. Despite the different proportions, we carefully reviewed the curated pairs and derived several findings. ① Humans are more likely to reuse existing roles if the provided source code is not lengthy. Hence, the pairs are mostly aligned for cases with existing roles defined in the source code. ② Humans tend to reuse function names as permissions without distilling them into abstract permissions as in our RBAC taxonomy. This phenomenon is especially evident for junior experts. It might indicate that humans require training and experience to accurately identify and summarize the proper role-permission pairs for correct patches. ③ Humans are inclined to give conservative roles, such as `owner`, `admin`, and *authorized user*. On average, 96.4 (81.69%) and 79.0 (66.95%) of the roles given by junior and senior experts respectively were conservative, contrasting with the 64 (54.24%) returned by ACFix.

These experts were further asked to draft patches for 6 failed cases by ACFix. After validating their patches with the same metrics, it turned out that half of their patches were correct according to all metrics. We took *Quixotic* [85] as an example to demonstrate the difference. Its brief vulnerability description is *Quixotic checks only the buyer's signature.* The vulnerable function `fillSellOrder` has multiple arguments and only checks the buyer's identity. Human experts were able to construct patches involving the AC checks against `buyer` as well as other necessary argument checks, such as `expiration` and `price`, according to their auditing experience. However, ACFix strictly included only the `buyer` role without flexibly involving other necessary checks.

Takeaway: ACFix's repairs are mostly aligned with those of humans and are finer-grained than those of both senior and junior experts. However, in rare cases (3/118), human experts are better at handling open issues based on their knowledge and experience without much guidance.

B. Fixes are Non-trivial to Devise by Humans

We further assess whether the 118 AC fixes attempted by ACFix are non-trivial to devise by humans based on the results of the surveyed second question. The results indicated that on average, junior and senior experts respectively identified 58.0 (49.15%) and 53.5 (45.34%) *NO* cases that are not straightforward to propose a patch. Based on majority voting, there were 42 (35.59%) *YES* cases in which most experts agreed on the straightforwardness of patch instrumentation. After manually inspecting them, we found they mostly (88%) belong to `initialization` and *changes of ownership*, which are straightforward to fix because only the *owner of the contract* should be checked. For the rest of the non-trivial cases, various role-permission pairs were included. For instance, `Mint`, which could be subject to abundant rules to implement the AC policy, is not straightforward to fix. Another example is the `Guardian` role [86] for ERC20. In this case, the original contract has three roles, `Guardian`, `Governor`, and `Minter`, which already form a hierarchy. To provide a proper fix, the practitioner must understand the hierarchy and the functionality of the target function, which could be a laborious and complicated task. These non-trivial cases could be effectively tackled by ACFix given that it has learned various rules and understood the hierarchical relationships of existing RBAC.

Case Study. We selected a case successfully repaired by ACFix, which was agreed by all participants to be tough to fix, to demonstrate ACFix's advantage. The case is the motivating example in Fig. 1 [41]. The fix is complicated because one has to understand the implicit design of the *external* function `depositFromOtherContract`. We first explain the reason behind the correct patch: As there is already a `deposit` function to deposit one's own values belonging to `msg.sender`, the vulnerable function `depositFromOtherContract` takes in an argument `_from` to deposit values on behalf of other users. However, depositing on behalf of others requires a trustworthy authority to act as a centralized agency. There are two trusted addresses defined in this contract, namely, *owner* and *bank*. Given that *bank* is set by *owner*, the proper role within the context to manage deposits is *bank*, which is exactly how the original author fixed it.

From the above case, we derive several challenges of manual patching: (1) Going through existing functions and distinguishing them from each other regarding the desired functionality, i.e., `depositFromOtherContract` and `deposit`; (2) Understanding the existing RBAC hierarchy based on the implementation of the chain of trust, i.e., *owner* and *bank*; (3) Understanding the implicit relationship between the design of a centralized agency and the existing role *bank*. In response to these challenges, ACFix could effectively mine the existing RBAC roles and implementations of the two existing deposit functions. Then, GPT-4 could understand the implicit logic between them to address challenges (2) and (3).

Takeaway: Around half of the AC fixes are non-trivial to devise by humans, indicating that ACFix can provide a unique complement to assist human-in-the-loop repair as a copilot. Through a case study, we conclude that with the aid of an LLM, the implicit logic can be dissected and streamlined from the source code, which is imperative for generating proper patches for AC vulnerabilities.

IX. Threats of Validity

Internal Threats: The primary threat to ACFix is the precision of static analysis. As ACFix mostly relies on Antlr to resolve dependency relationships of code statements using AST, rather than IR, ACG may not achieve high precision and recall. However, this potential inaccuracy does not markedly affect ACFix's capabilities for two reasons. First, the selection of role-permission pairs primarily depends on GPT-4's logical reasoning capabilities, provided there is sufficient context to infer role and permission. Second, in most cases, ACFix performs static analysis within a single contract file. This means that most of the call graphs, def-use chains of state variables, and mappings between parameters of functions could reliably rely on name mappings. Therefore, the static analysis in ACFix may be flawed, but it suffices to support context understanding of GPT-4.

An internal threat to validity stems from the dataset used for evaluating AC vulnerabilities, which was gathered primarily from limited online sources, specifically DefiHackLabs [70] and tintinweb [71], resulting in unequal representation and potential incompleteness. Such imbalanced distributions across data sources might inadvertently introduce biases. To mitigate this issue, we designed RQ4 explicitly to analyze the performance of ACFix and baseline methods within individual categories, thereby reducing sensitivity to data imbalance. Additionally, we made considerable efforts to include as many cases as possible from diverse Internet sources, enlarging the dataset to facilitate a more comprehensive and fair comparison.

The last external threat to validity lies in the interpretability of LLMs like GPT-4, which ACFix relies on. Due to their black-box nature, LLMs may generate outputs that are difficult to explain or verify, potentially introducing incorrect or inconsistent repairs. Issues such as hallucination, prompt sensitivity, and lack of transparency in reasoning pose risks, particularly in security-critical contexts like access control. To mitigate these concerns, we constrain the LLM's output space using a dynamic taxonomy of RBAC role-permission pairs, reducing the likelihood of invalid predictions. Additionally, we integrate static analysis checks to validate the syntactic and semantic correctness of generated patches. We also employ carefully crafted in-context prompts to enhance stability and reduce variation across similar cases. Furthermore, ACFix incorporates a dual-agent feedback framework, where a validation agent assesses the generated output and provides feedback to the generator, enabling iterative refinement. Together, these mechanisms help enhance interpretability and reliability, though we acknowledge that challenges inherent to LLMs remain an open research issue.

External Threat: The potential threat to validity arises from the initial reliance on a manually curated RBAC taxonomy. Due to inherent constraints in the available dataset, this taxonomy may not comprehensively represent all possible role-permission relationships encountered in real-world smart contract implementations. Such incompleteness could potentially lead to inaccuracies or misidentifications of role-permission pairs during vulnerability repair. To address this limitation, we incorporated an adaptive mechanism within our approach, enabling the automatic addition of newly identified role-permission pairs to dynamically expand the taxonomy. This strategy effectively mitigates the risks posed by a static, finite taxonomy, ensuring greater robustness and adaptability of the proposed solution.

X. Related Work

A. Smart Contract Repair

Smart contract vulnerability repair has seen significant progress, such as Aroc [18], SmartShield [16], SGuard [15], Elysium [19], SCRepair [17], and SmartFix [14]. However, research on repairing AC-related vulnerabilities remains limited. Tools like Aroc and SmartShield do not support AC repairs, SGuard addresses only *tx.origin* misuse, and Elysium fixes only two sensitive operations. SCRepair's effectiveness is constrained by manual unit tests, while SmartFix handles only *tx.origin* and *re-initialization* vulnerabilities.

In light of the above, ACFix stands out in two ways: ① Human-Level Reasoning: We address and resolve the limitations inherent in prior works that relied solely on predefined templates. By utilizing GPT-4, our method engages in conversational sessions employing CoT and MAD, which allows ACFix to achieve human-like reasoning. This marks a significant advancement in the methodology for AC vulnerability repairs. ② Comprehensive Coverage: Many existing tools support AC vulnerabilities but are often restricted to handling conventional patterns. In contrast, ACFix boasts the capability to address AC vulnerabilities across diverse scenarios.

B. Traditional Program Repair

Numerous works have focused on repairing bugs or vulnerabilities in traditional software [42], [87], [88], [89], [90], [91], [92], [93], [94], [95], [96], [97], [98], especially in C [91], [92], [93], Java [90], [94], and PHP [87]. Moreover, several concurrent works [42], [88], [89], [90], [99], [100] propose LLM-based APR solutions for bug fixes. For example, Xia et al. [88] studied the effectiveness of LLMs for APR and found that LLMs generally outperform traditional approaches. ChatRepair [89] employs multiple sessions for interactive repair with GPT-4. Repilot [90] innovatively combines the completion engine with LLM to synergistically generate patches. FitRepair [42] leverages the *plastic surgery hypothesis* to repair bugs using existing code ingredients by performing static analysis and information retrieval on the source code. Other related works [91], [92], [93], [94], [95], [96], [97], [98] mostly employ traditional methods, such as Neural Machine Translation, to synthesize repairs for bugs or iteratively search for proper patches. Our work shares several common practices, such as conversational sessions and existing ingredient reuse, but uniquely mines RBAC practices and relevant code context to guide LLMs.

XI. Future Work

Building upon the insights and infrastructure established by ACFix, we identify several promising directions for future exploration:

- **LLM-guided Repair of Advanced Vulnerabilities.** While this work focuses on AC vulnerabilities, our approach establishes a foundation for addressing a broader class of complex smart contract vulnerabilities. In future work, we plan to extend ACFix's reasoning and repair capabilities to additional vulnerability types that require deep semantic understanding, such as reentrancy with indirect triggers, improper state transitions, delegatecall misuse, and incorrect payment logic. These categories often involve non-trivial control flow, cross-contract dependencies, or subtle logic flaws that static patterns alone cannot effectively capture. Enhancing ACFix with formal specifications, symbolic reasoning, or integration with domain-specific ontologies may further improve its adaptability and accuracy.
- **Detection and Repair of Multi-Function AC Vulnerabilities.** A more advanced but rare class of AC vulnerabilities involves multiple functions collectively contributing to unauthorized privilege escalation. These vulnerabilities are especially challenging, as they may not exhibit direct or transitive call relationships but instead share critical state variables that facilitate cross-function interactions. We intend to investigate this class of vulnerabilities by modeling state-dependent attack surfaces and designing analysis techniques to identify latent privilege escalation paths that span disjointed code regions. This form of vulnerability is rare but significantly harder to detect and mitigate.
- **Adaptive Taxonomy Evolution via Online Learning.** While ACFix leverages a dynamic taxonomy mined from a large corpus of on-chain contracts, smart contract development practices continue to evolve, introducing new roles, patterns, and access semantics. This new emerging knowledge may not be accommodated well by merely updating the taxonomy with more RBAC pairs. To maintain robustness and adaptability, we envision extending the current static RBAC taxonomy into a dynamic, Retrieval-Augmented Generation (RAG) system. In this setting, the LLM would query an updatable knowledge base of role-permission pairs—continuously refined from emerging contracts, validator feedback, and user interaction logs—enabling it to incorporate the latest access control practices

during repair. This dynamic integration would reduce reliance on static assumptions, enhance coverage of long-tail or novel RBAC cases, and provide a pathway for ACFix to generalize beyond its original training distribution with minimal human intervention.

- **LLM-guided AC Vulnerability Detection.** With the mined RBAC taxonomy and enhanced context comprehension mechanisms, we plan to extend the scope of ACFix from repair to detection. By leveraging LLMs' ability to semantically understand AC context, although ACFix has advanced the usage of LLM on smart contract security repair, we aim to further detect improper RBAC implementation by identifying role-permission mismatches relative to the intended functionality of contracts. However, vulnerability detection presents unique challenges compared to repair, particularly in terms of scalability. Unlike repair, which starts from known vulnerable functions, detection must assume that any function could be misconfigured and thus requires comprehensive analysis across the entire contract. This substantially increases computational costs, as LLMs must perform intricate RBAC reasoning for all functions. To address this, we plan to incorporate static filtering techniques to preselect likely-vulnerable candidates, enabling scalable and efficient LLM-guided detection without exhaustive analysis.
- **Advanced Validation Paradigm.** We plan to further advance the validation process by exploring more sophisticated MAD frameworks. In particular, we aim to incorporate additional specialized agents, such as adversarial critics and domain-specific oracles, to enrich the debate dynamics and improve the reliability of patch validation. We also intend to investigate adaptive debate strategies, where the validation process dynamically adjusts the roles or number of agents based on the complexity of the repair task. These directions are expected to enhance both the robustness and explainability of the validation stage, and will be integrated into future iterations of ACFIX.

XII. CONCLUSION

This paper proposed ACFIX for repairing AC vulnerabilities in smart contracts by guiding LLMs with AC practices and code context. We developed an RBAC taxonomy from on-chain contracts and a slicing algorithm to extract AC-related context. Equipped with check rules and *validator* of MAD, ACFIX repaired 94.92% of cases in our dataset, outperforming existing tools. Our evaluation included a human study to assess the quality of ACFIX's repairs compared to humans'.

ACKNOWLEDGMENT

This work was done while Daoyuan Wu and Ye Liu were at NTU.

OPEN SCIENCE POLICY

To facilitate replication and future research, we have released our source code and dataset on an anonymous website [75].

REFERENCES

[1] G. Wood et al., "Ethereum: A secure decentralised generalised transaction ledger," Ethereum Project Yellow Paper, 2014, pp. 1–32.

[2] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. Knottenbelt, "Sok: Decentralized finance (DeFi)," in *Proc. 4th ACM Conf. Adv. Financial Technol. (AFT)*, 2023, pp. 30–46.

[3] D. Das, P. Bose, N. Ruaro, C. Kruegel, and G. Vigna, "Understanding security issues in the NFT ecosystem," in *Proc. ACM CCS (CCS),* New York, NY, USA: ACM, 2022, pp. 667–681, doi: 10.1145/3548606.3559342.

[4] Solidity. [Online]. Available: https://soliditylang.org/

[5] 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," 2023, *arXiv:2303.13770*.

[6] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proc. NDSS*, 2018, pp. 1–12.

[7] P. Daian et al., "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *Proc. IEEE Symp. Secur. Privacy (SP),* Piscataway, NJ, USA: IEEE Press, 2020, pp. 910–927.

[8] S. Wu et al., "DeFiRanger: Detecting price manipulation attacks on DeFi applications," 2021, *arXiv:2104.15068*.

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

[10] "Parity wallet attack." OpenZeppelin. [Online]. Available: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7

[11] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities," in *Proc. 41st ACM PLDI*, 2020, pp. 454–469.

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

[13] Y. Fang et al., "Beyond 'protected' and 'private': An empirical security analysis of custom function modifiers in smart contracts," in *Proc. ACM ISSTA*, 2023, pp. 1157–1168.

[14] S. So and H. Oh, "SmartFix: Fixing vulnerable smart contracts by accelerating generate-and-verify repair using statistical models," in *Proc. 31th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2023, pp. 185–197.

[15] T. D. Nguyen, L. H. Pham, and J. Sun, "SGUARD: Towards fixing vulnerable smart contracts automatically," in *Proc. IEEE Symp. Secur. Privacy (SP)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 1215–1229.

[16] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "SMARTSHIELD: Automatic smart contract protection made easy," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. ReEng. (SANER).* Piscataway, NJ, USA: IEEE Press, 2020, pp. 23–34.

[17] X. L. Yu, O. Al-Bataineh, D. Lo, and A. Roychoudhury, "Smart contract repair," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 29, no. 4, pp. 1–32, 2020.

[18] H. Jin, Z. Wang, M. Wen, W. Dai, Y. Zhu, and D. Zou, "Aroc: An automatic repair framework for on-chain smart contracts," *IEEE Trans. Softw. Eng.*, vol. 48, no. 11, pp. 4611–4629, Nov. 2022.

[19] C. F. Torres, H. Jonker, and R. State, "Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts," in *Proc. 25th Int. Symp. Res. Attacks, Intrusions Defenses*, 2022, pp. 115–128.

[20] M. Rodler, W. Li, G. O. Karame, and L. Davi, "{EVMPatch}: Timely and automated patching of Ethereum smart contracts," in *Proc. 30th USENIX Secur. Symp. (USENIX Secur.),* 2021, pp. 1289–1306.

[21] "Smart contract initialization." Cyberark. [Online]. Available: https://www.cyberark.com/resources/threat-research-blog/how-to-write-a-poc-for-an-uninitialized-smart-contract-vulnerability-in-badgerdao-using-foundry

[22] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," 2018, *arXiv:1802.06038*.

[23] "Unchecked low-level call." Medium. [Online]. Available: https://simonbusch.medium.com/smart-contracts-vulnerability-explained-unchecked-send-ed8b5606813a

[24] R. S. Sandhu, "Role-based access control," in *Advances in Computers*, vol. 46. New York, NY, USA: Elsevier, 1998, pp. 237–286.

[25] H. Touvron et al., "Llama 2: Open foundation and fine-tuned chat models," 2023, *arXiv:2307.09288*.

[26] OpenAI, "GPT-4 technical report," 2023*, arXiv:2303.08774*.

[27] H. Tian et al., "Is ChatGPT the ultimate programming assistant – How far is it?" 2023*, arXiv:2304.11938*.

[28] "ChatGPT hallucination." Fortune. [Online]. Available: https://fortune.com/2023/08/01/can-ai-chatgpt-hallucinations-be-fixed-experts-doubt-altman-openai/

[29] T. Liang et al., "Encouraging divergent thinking in large language models through multi-agent debate," 2023, *arXiv:2305.19118*.

[30] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in *Proc. Adv. Neural Inf. Process. Syst.*, 2022, vol. 35, pp. 24824–24837.

[31] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019. [Online]. Available: https://arxiv.org/abs/1810.04805

[32] A. Radford et al., "Improving language understanding by generative pre-training." [Online]. Available: https://www.mikecaptain.com/resources/pdf/GPT-1.pdf

[33] "ChatGPT." Open AI. [Online]. Available: https://chat.openai.com/

[34] "GPT3.5." Open AI. [Online]. Available: https://platform.openai.com/docs/models

[35] A. Q. Jiang et al., "Mistral 7b," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[36] "Llama3." Hugging Face. [Online]. Available: https://huggingface.co/meta-llama/Meta-Llama-3-8B

[37] N. Szabo, "Smart contracts: Building blocks for digital markets," 1997. [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html

[38] "ERC 20," Ethereum, 2024. [Online]. Available: https://ethereum.org/en/developers/docs/standards/tokens/erc-20/

[39] "OpenZepplin Access Control," GitHub. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol

[40] "GYMNetwork attack." Accessed: Aug. 26, 2023. [Online]. Available: https://wooded-meter-1d8.notion.site/Incorrect-access-control-579d6806099e4304aa761ade1d1c7e

[41] "Example address." BscScan. [Online]. Available: https://bscscan.com/address/0x0288fba0bf19072d30490a0f-3c81cd9b0634258a#code#F1#L291

[42] C. S. Xia, Y. Ding, and L. Zhang, "Revisiting the plastic surgery hypothesis via large language models," 2023, *arXiv:2303.10494*.

[43] T. Schick et al., "Toolformer: Language models can teach themselves to use tools," 2023, *arXiv:2302.04761*.

[44] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proc. NeurIPS*, 2020, pp. 9459–9474.

[45] J. Baek, A. F. Aji, and A. Saffari, "Knowledge-augmented language model prompting for zero-shot knowledge graph question answering," 2023, *arXiv:2306.04136*.

[46] Z. Wang, S. Mao, W. Wu, T. Ge, F. Wei, and H. Ji, "Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration," 2023, *arXiv:2307.05300*.

[47] Y. Du, S. Li, A. Torralba, J. B. Tenenbaum, and I. Mordatch, "Improving factuality and reasoning in language models through multiagent debate," 2023, *arXiv:2305.14325*.

[48] Z. Xi et al., "The rise and potential of large language model based agents: A survey," 2023, *arXiv:2309.07864*.

[49] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[50] "Ethereum contracts." GitHub. [Online]. Available: https://github.com/tintinweb/smart-contract-sanctuary-ethereum

[51] L. Liu, L. Wei, W. Zhang, M. Wen, Y. Liu, and S. Cheung, "Characterizing transaction-reverting statements in Ethereum smart contracts," in *Proc. 36th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2021, pp. 630–641.

[52] "Open card sorting." Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Card_sorting

[53] I. David, L. Zhou, K. Qin, D. Song, L. Cavallaro, and A. Gervais, "Do you still need a manual smart contract audit?" 2023, *arXiv:2306.12338.*

[54] Y. Sun et al., "LLM4Vuln: A unified evaluation framework for decoupling and enhancing LLMs' vulnerability reasoning," 2024, *arXiv:2401.16185*.

[55] Z. Yuan et al., "No more manual tests? Evaluating and improving ChatGPT for unit test generation," 2023, *arXiv:2305.04207*.

[56] Y. Sun et al., "GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng. (ICSE)*, 2024, pp. 1–13.

[57] C. Chen et al., "When ChatGPT meets smart contract vulnerability detection: How far are we?" 2023, *arXiv:2309.05520*.

[58] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, no. 4, pp. 352–357, Jul. 1984.

[59] A. De Lucia, A. R. Fasolino, and M. Munro, "Understanding function behaviors through program slicing," in *Proc. 4th Workshop Program Comprehension (WPC)*, Piscataway, NJ, USA: IEEE Press, 1996, pp. 9–18.

[60] S. Badihi, F. Akinotcho, Y. Li, and J. Rubin, "ARDiff: Scaling program equivalence checking via iterative abstraction and refinement of common code," in *Proc. 28th ACM FSE*, 2020, pp. 13–24.

[61] S. Badihi, K. Ahmed, Y. Li, and J. Rubin, "Responsibility in context: On applicability of slicing in semantic regression analysis," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE),* Piscataway, NJ, USA: IEEE Press, 2023, pp. 563–575.

[62] Y. Xiao et al., "{MVP}: Detecting vulnerabilities using {patch-enhanced} vulnerability signatures," in *Proc. 29th USENIX Secur. Symp. (USENIX Secur.),* 2020, pp. 1165–1182.

[63] L. Zhang et al., "Has my release disobeyed semantic versioning? Static detection based on semantic differencing," in *Proc. 37th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2023, pp. 1–12.

[64] X. Yi, Y. Fang, D. Wu, and L. Jiang, "BlockScope: Detecting and investigating propagated vulnerabilities in forked blockchain projects," in *Proc. ISOC NDSS*, 2023.

[65] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

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

[67] T. J. Parr and R. W. Quong, "ANTLR: A predicated-*LL* (*k*) parser generator," *Softw., Pract. Exp.*, vol. 25, no. 7, pp. 789–810, 1995.

[68] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 530–541.

[69] "National vulnerability database," NIST, 2023. [Online]. Available: https://nvd.nist.gov/

[70] "DeFi hack labs." GitHub. [Online]. Available: https://github.com/SunWeb3Sec/DeFiHackLabs

[71] "Tintinweb Vul Dataset." GitHub. [Online]. Available: https://github.com/tintinweb/smart-contract-vulndb/tree/main

[72] "Blocksec building blockchain security infrastructure." BlockSec. Accessed: Sep. 1, 2023. [Online]. Available: https://blocksec.com/#blogs

[73] SlowMist. Accessed: Sep. 1, 2023. [Online]. Available: https://www.slowmist.com/ (accessed Sep. 1, 2023).

[74] Medium. Accessed: Sep. 1, 2023. [Online]. Available: https://medium.com/

[75] "ACFix Website." Google. [Online]. Available: https://sites.google.com/view/acfixsmartcontract

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

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

[78] C. Gao, W. Yang, J. Ye, Y. Xue, and J. Sun, "sGuard+: Machine learning guided rule-based automated vulnerability repair on smart contracts," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 5, pp. 1–55, 2024.

[79] J.-R. Giesen, S. Andreina, M. Rodler, G. O. Karame, and L. Davi, "Practical mitigation of smart contract bugs," 2022, *arXiv:2203.00364*.

[80] C. Wang, J. Zhang, J. Gao, L. Xia, Z. Guan, and Z. Chen, "ContractTinker: LLM-empowered vulnerability repair for real-world smart contracts," in *Proc. 39th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2024, pp. 2350–2353.

[81] V. Mothukuri, R. M. Parizi, and J. L. Massa, "LLMSmartsec: Smart contract security auditing with LLM and annotated control flow graph," in *Proc. IEEE Int. Conf. Blockchain (Blockchain),* Piscataway, NJ, USA: IEEE Press, 2024, pp. 434–441.

[82] "OpenAI Formatting." Open AI. [Online]. Available: https://platform.openai.com/docs/api-reference/audio/createTranscription#audio-createtranscription-response_format

[83] Etherscan. [Online]. Available: https://etherscan.io/

[84] "How much does GPT-4 cost?" Open AI. [Online]. Available: https://help.openai.com/en/articles/7127956-how-much-does-gpt-4-cost

[85] "Quitoxic." GitHub. [Online]. Available: https://github.com/SunWeb3Sec/DeFiHackLabs#20220701-quixoti–coptimism-nft-marketplace

[86] "Guardian role for ERC20." GitHub. [Online]. Available: https://github.com/sherlock-audit/2023-04-unitasprotocol-judging/blob/main//false/126.md

[87] S. Son, K. S. McKinley, and V. Shmatikov, "Fix me up: Repairing access-control bugs in web applications," in *Proc. NDSS*, 2013.

[88] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE),* New York, NY, USA: ACM, 2023, pp. 1482–1494.

[89] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," 2023, *arXiv:2304.00385*.

[90] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," 2023, *arXiv:2309.00608.*

[91] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "VulRepair: A T5-based automated software vulnerability repair," in *Proc. 30th ACM FSE*, 2022, pp. 935–947.

[92] J. Chi, Y. Qu, T. Liu, Q. Zheng, and H. Yin, "SeqTrans: automatic vulnerability fix via sequence to sequence learning," *IEEE Trans. Softw. Eng.*, vol. 49, no. 2, pp. 564–585, Feb. 2023.

[93] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *IEEE Trans. Softw. Eng.*, vol. 49, no. 1, pp. 147–165, Jan. 2023.

[94] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 298–309.

[95] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 802–811.

[96] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 166–178.

[97] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proc. 43rd Annu. ACM POPL*, 2016, pp. 298–312.

[98] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. IEEE 31st Int. Conf. Softw. Eng.,* Piscataway, NJ, USA: IEEE Press, 2009, pp. 364–374.

[99] K. Huang et al., "An empirical study on fine-tuning large language models of code for automated program repair," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2023, pp. 1162–1174.

[100] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 1430–1442.