Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

"Enhancing Smart Contract Security: Leveraging Generative AI for Vulnerability Remediation"

[1]Sangeetha R, [2]Dr. Veena M N
[1]*Assistant Professor CIST, ManasaGangortri, University of Mysore Mysuru-570006*
sangeethauom9@gmail.com Orcid Id:0009-0004-1452-0549
[2]*Professor Department of MCA PES college of Engineering, Shankar Gowda road, Mandya-571401*
veenadisha1@gmail.com Orcid Id:0009-0004-5953-1065

ARTICLE INFO

Received: 28 Dec 2024

Revised: 18 Feb 2025

Accepted: 26 Feb 2025

ABSTRACT

The rapid proliferation of decentralized applications utilizing blockchain technology has emphasized the crucial role of smart contracts in ensuring secure and reliable transaction execution. Nonetheless, the inherent complexities related to smart contract development have resulted in various vulnerabilities, leading to significant financial losses and a deterioration of trust in the ecosystem. While traditional vulnerability detection techniques have made strides, the dynamic nature of smart contract vulnerabilities necessitates the exploration of novel, adaptive approaches. This paper proposes a generative AI-driven methodology for enhancing smart contract security by leveraging language models finetuned on curated datasets of solidity code, vulnerabilities, and remediation strategies. By harnessing the power of generative AI, our methodology effectively incorporates the identification of vulnerabilities and their subsequent resolution, offering aholistic approach to enhancing the security of smart contracts. While prior work has explored leveraging large language models for automatic code repair in other programming languages, this remains an open challenge for Solidity smart contracts. This study enhances the field of smart contract security by presenting an innovative resolution that merges AI methodologies with blockchain technology, with the goal of mitigating risks and fostering trust in decentralized systems.

Keywords: Blockchain, Smart Contracts, Vulnerability Detection, Vulnerability Remediation, Generative AI, Language Models, Solidity

I. INTRODUCTION

Blockchain technology is a groundbreaking innovation that underpins a vast array of decentralized applications and digital assets. At its core, blockchain operates as a decentralized and distributed ledger, which ensures that all transactions and data entries are recorded across multiple nodes in a network. This decentralized nature eliminates the need for a central authority, thereby enhancing security and trust among participants. Each transaction or data entry is grouped into blocks, and these blocks are cryptographically linked to form a chain, hence the term" blockchain." This structure ensures that once data is recorded, it cannot be altered or tampered with without the consensus of the network, making the blockchain immutable.

Blockchain technology has seen widespread adoption across various industries, fundamentally transforming how businesses operate and driving substantial growth in popularity. In finance [1], [2], blockchain has revolutionized the way transactions are conducted by enabling peerto-peer transfers without the need for intermediaries, thereby reducing costs and increasing transaction speed. Cryptocurrencies like Bitcoin [3] and Ethereum are prime examples of blockchain's impact, offering new forms of digital assets and investment opportunities. In the supply chain sector [4], [5], blockchain enhances transparency and traceability, allowing companies to track products from production to delivery, thus ensuring authenticity and reducing fraud. This is particularly beneficial in industries like food and pharmaceuticals, where provenance and quality control are critical. Healthcare [6], [7] has also been significantly impacted, with blockchain providing secure and immutable records of patient data, facilitating better data sharing and ensuring patient privacy. Moreover, blockchain has found applications in real estate, enabling seamless and transparent property transactions, and in voting systems, where it enhances the integrity and security of elections. The technology's ability to provide secure, transparent, and tamper-proof records has led to its growing popularity, with businesses and governments worldwide exploring its potential to create more efficient, trustworthy, and decentralized systems. This surge in interest is reflected in the increasing number of blockchain startups, the rising market capitalization of cryptocurrencies, and the integration of blockchain solutions by major corporations, underscoring its transformative impact and promising future.

Smart contracts, while revolutionary in their ability to automate and enforce agreements on blockchain platforms, are vulnerable to various types of exploits that can compromise their integrity and security. These vulnerabilities can be broadly categorized into several types, each posing unique risks to the proper functioning of smart contracts. Common vulnerabilities include reentrancy, access control issues, arithmetic problems such as integer overflow and underflow, unchecked return values for low-level calls, denial of service attacks, and vulnerabilities related to timing manipulation and transaction order dependency.

Reentrancy vulnerabilities occur when a contract's method can be called repeatedly before the first invocation completes, potentially allowing an attacker to manipulate the contract's state unexpectedly. Access control issues arise when improper permissions or conditions allow unauthorized parties to execute functions they should not have access to. Arithmetic issues like integer overflow and underflow occur when mathematical operations exceed the limits of data types, leading to unintended behavior that can be exploited. Denial of service attacks can cripple a contract's functionality by overwhelming it with requests, while timing manipulation vulnerabilities allow attackers to exploit discrepancies in timestamps for malicious purposes. These vulnerabilities underscore the importance of rigorous security audits and continuous monitoring to safeguard smart contracts from exploitation and uphold the trust placed in blockchain technology.

Notable incidents, such as the DAO attack on the Ethereum platform, where over $70 million was drained due to a reentrancy vulnerability [8], have underscored the urgency to fortify smart contract security. Traditional approaches to vulnerability detection and mitigation have made strides, but the ever-evolving landscape of threats necessitates more adaptive and robust solutions. Moreover, while existing static analysis techniques can effectively identify vulnerabilities without executing the code, they often grapple with significant computational resource requirements and scalability constraints, especially when dealing with larger or more intricate smart contracts. Addressing these limitations is pivotal to ensuring the security and resilience of real-world decentralized applications.

Emerging at the forefront of artificial intelligence, generative AI and large language models (LLMs) have demonstrated remarkable prowess in various domains, including code generation and repair. Their ability to learn intricate patterns and generate contextually relevant output holds immense potential for streamlining vulnerability detection and remediation in smart contracts. This work addresses these challenges by introducing a two-step, generative AI-based solution leveraging fine-tuned LLMs for automated vulnerability detection and remediation in Solidity smart contracts. This approach offers a scalable, adaptive alternative to traditional static analysis tools.

A. Motivation

The immutable nature of deployed smart contracts, coupled with the substantial financial risks posed by vulnerabilities, necessitates robust and adaptive security measures. The existing static analysis techniques face scalability and resource limitations. This research harnesses the power of generative AI and large language models to bridge this gap, providing a cutting-edge solution for seamless vulnerability detection and remediation in smart contracts.

B. Main Contributions of the paper:

- Develop a novel framework that integrates generative AI and LLMs for smart contract vulnerability detection and remediation
- Empirically evaluate the effectiveness of the proposed approach across diverse vulnerability classes
- Demonstrate the superiority of the generative AI-driven solution over traditional static analysis techniques
- Provide insights into the practical implementation and deployment of the developed framework

II. RELATED WORK

A. Smart Contracts and Vulnerabilities

Smart contracts, introduced by Nick Szabo in the early 1990s [9], represent a significant advancement in digital agreements. These self-executing contracts with the terms of the agreement directly written into code have found their most prominent implementation on the Ethereum platform. Launched in 2015 [10], Ethereum revolutionized the blockchain landscape by offering a programmable blockchain that supports the creation and execution of smart contracts through its specialized language, Solidity [11].

The versatility of smart contracts has led to their adoption in various applications, from decentralized finance (DeFi) to non-fungible tokens (NFTs). By eliminating intermediaries and automating agreement execution, smart contracts enhance transparency, reduce costs, and increase efficiency in numerous sectors [12]. The Ethereum Virtual Machine (EVM) [12] plays a crucial role in this ecosystem, executing contract code and storing results on the blockchain, while Ether serves as both an incentive for network participants and a means to pay for computational work.

However, the immutable nature of deployed smart contracts, combined with their often high-value transactions, makes them attractive targets for malicious actors. Vulnerabilities in smart contract code can lead to severe consequences, as demonstrated by the infamous DAO hack in 2016, which resulted in a loss of over 70 million dollars [8]. Common vulnerabilities include reentrancy attacks, integer overflow/underflow issues, gas limit problems, timestamp dependence, and susceptibility to denial-of-service attacks [13]. These security risks underscore the critical need for robust vulnerability detection and mitigation strategies in smart contract development and deployment.

B. Vulnerability Detection and Mitigation Techniques

The detection and mitigation of smart contract vulnerabilities have become crucial in ensuring the security and integrity of blockchain-based applications. Researchers and developers have employed a wide array of techniques to address this challenge, ranging from traditional static analysis methods to advanced machine learning approaches for detection, and various repair strategies for mitigation.

In terms of detection, traditional approaches rely on techniques such as bytecode and source code analysis. Tools like Oyente, Osiris, and Mythril [13] exemplify this category, employing various analysis techniques to identify potential security flaws in smart contracts. More recently, machine learning-based methods have gained traction. TP-Detect [15], [24], for instance, utilizes trigram feature

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

TABLE I COMPARISON OF VULNERABILITY DETECTION TOOLS

Tool	Analysis Level	Vulnerabilities	Evaluation Techniques
A new scheme of vulnerability analysis in smart contract with machine learning [14]	Bytecode	Has short address, has flows, is greedy	Average F1 scores NN=76.6% CNN=83.3% RF=90.3%
TP-Detect [15]	Opcode	Integer flow, Call Stack Depth, Reentrancy, Parity Multisig, Timestamp, Transaction-Ordering Dependency	Average F1 scores RF=96.66% NB=99.41% kNN=90.75%
Attention-based Machine Learning Model for Smart Contract Vulnerability Detection [16]	Bytecode	Reentrancy, Arithmetic Issues, Time manipulation	Average F1 Score =87.66%
ESCORT [17]	Bytecode	Call stack Depth, Reentrancy, Multiple Sends, Accessible selfdestruct, DoS (Unbounded Operation), Tainted selfdestruct, Money concurrency, Assert violation	Average F1 score=95%
Smart Contract Vulnerability Detection Based on Deep Learning and Multimodal Decision Fusion[18]	Source code, Opcode, Flow GraphControl	Arithmetic Issues, Reentrancy,Transaction-Ordering, Dependency,Locked-Ether	Average F1 Score =67.125%
Dynamit [19]	Transaction metadata	Reentrancy	RF=89% NB=83% kNN=75%
Eth2Vec [20]	EVM bytecode, Assembly code,Abstract Syntax Trees	Reentrancy, Transaction-Ordering Dependence, Timestamp Dependency, Callstack depth, Unchecked call etc	Average F1 score=57.50%
ContractWard [21]	Opcode	Reentrancy, Callstack Depth,Timestamp, Dependency,Transaction-Ordering Dependence,Integer flow	Average F1 score=96%
Hunting the Ethereum Smart Contract: Color-inspired Inspection ofPotential Attacks [22]	Bytecode	ZeroFunctionSelector,SendFailsForZeroEther,SkipEmptyStringLiteral,Delegate CallReturn, Value,ConstantOptimizerSubtraction etc	Average F1 score=97%
SmartCheck [23]	Source code	Reentrancy, Unchecked external call, Timestamp dependency, DoS by external contract etc	FDR=68.97% FNR=47.06%

extraction and pixel values to detect vulnerabilities in Ethereum smart contracts. Other frameworks incorporate deep neural networks and transfer learning [17] to enhance detection capabilities. Specialized techniques have also emerged, such as attention-based models [16] and multimodal decision fusion [18], which aim to improve accuracy and efficiency in vulnerability detection.

Mitigation techniques for smart contract vulnerabilities can be broadly categorized into off-chain and on-chain repairs [25]. Off-chain repair tools like SCRepair [26] use genetic algorithm-based approaches to modify complex smart contracts at the source code level. SGUARD [27] employs a two-step process involving symbolic execution traces and static analysis to identify and fix vulnerabilities. ContractFix [28] introduces a framework for automatically generating source code patches, demonstrating a 94% success rate in fixing detected vulnerabilities. On-chain repair solutions, such as Aroc [25], provide protection for already deployed contracts without altering their source code. EVMPATCH [29] offers a unique approach by combining bytecode rewriting with a proxy-based upgradeable smart contract method, focusing on addressing access control errors and integer bugs. These diverse mitigation strategies offer developers and blockchain networks various options to enhance smart contract security, each with its own strengths and limitations.

While the success of static and dynamic analysis tools is certain, these techniques usually do not generalize to novel vulnerabilities. Generative AI techniques, especially instruction-tuned LLMs, offer a potential way forward through learning contextual representations of secure patterns in code. Few have utilized them for remediation in smart contracts, though, creating the void this paper seeks to bridge.

C. Generative AI and Automated Code Repair

The application of generative AI to automated code repair, including smart contract vulnerability remediation, represents a cutting-edge approach to enhancing software security. This field has seen rapid advancements, with various techniques demonstrating promising results in addressing complex coding issues automatically.

One notable approach in this domain is NEVERMORE [31], which generates bug fixes that closely resemble human-written patches. In empirical evaluations, 21.2% of the fixes generated by NEVERMORE matched humanwritten solutions, showcasing its effectiveness in addressing real-world bugs. Moreover, this approach has outperformed several state-of-the-art deep learning Automated Program Repair (APR) techniques in fixing previously unresolved bugs, highlighting its potential for improving smart contract security.

TABLE II COMPARISON OF SMART CONTRACT REPAIR TOOLS

Framework	Analysis Level	Pros	Cons
SCRepair [26]	Source Code	Genetic algorithms provide effective fixes	Resource intensive, can't be scaled for complex programs
SGUARD [27]	Source Code	Fixes do not introduce significant overhead w.r.t execution speed and gas consumption	Path explosion, resource intensive
CONTRACTFIX [28]	Source Code	Includes static verfication	Can potentially impact performance
EVMPATCH [29]	Bytecode	Integrated with multiple static analysis tools	Limited extendibility
SMARTSHIELD [30]	Bytecode	87.6% of the real-world contracts were fully rectified	Increased gas consumption
Aroc [25]	Source Code	It can fix deployed contracts	It can break the immutable nature of contracts

Recent developments have also seen the integration of Large Pre-Trained Language Models (PLMs) into APR frameworks. RING [32], for example, is a multilingual repair engine that leverages Codex, a large language model

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

trained specifically on code. This approach streamlines program repair through stages such as fault localization, code transformation, and candidate ranking, demonstrating efficacy across multiple programming languages. Similarly, InferFix [33] combines static analysis for bug detection with a fine-tuned large language model for program repair, showing successful practical applications in real-world development environments. Comparative studies [34] have further revealed that the direct application of PLMs can significantly outperform existing APR techniques across diverse repair datasets and programming languages, indicating a promising direction for future research in smart contract vulnerability remediation.

III. METHODOLOGY

The proposed methodology for enhancing smart contract security leverages Generative AI and Large Language Models (LLMs) to automatically detect and rectify vulnerabilities in smart contracts, addressing the limitations of existing repair tools. The complete process is illustrated in Fig. 1. It begins by fine-tuning the LLM on datasets containing smart contracts and their corresponding vulnerabilities and pairs of vulnerable and fixed smart contracts, allowing it to discern between the two classes and effectively identify vulnerabilities in new contracts. The process then takes a vulnerable smart contract as input and employs the trained LLM to generate contextually appropriate patches for the identified vulnerabilities. The tool's output includes a report detailing the detected vulnerabilities alongside the fortified smart contract, now patched and ready for deployment on the blockchain, thus significantly bolstering security and mitigating exploitation risks.

A. Overview of Technologies used

1) Transformers: We utilize the Hugging Face Transformers library, a popular and powerful toolkit for working with state-of-the-art natural language processing models. This library provides a unified interface for loading and fine-tuning pre-trained language models, as well as functionalities for tokenization, data processing, and model evaluation.

2) Bits and Bytes Configuration: To enable efficient inference and reduce memory footprint, we employ the Bits and Bytes Configuration, a quantization technique that represents model weights using lower-precision data types (e.g., 4-bit or 8-bit). This technique allows us to leverage hardware accelerators and optimize performance without significant accuracy loss.

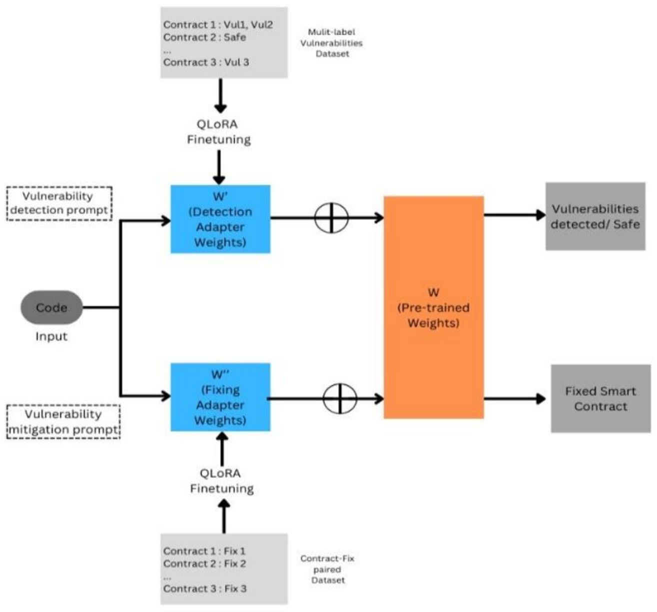

Contract 1 : Vul1, Vul2
Contract 2 : Safe
...
Contract 3 : Vul 3
Mulit-label Vulnerabilities Dataset
QLoRA Finetuning
Vulnerability detection prompt
W' (Detection Adapter Weights)
Code Input
W (Pre-trained Weights)
Vulnerabilities detected/ Safe
Vulnerability mitigation prompt
W'' (Fixing Adapter Weights)
Fixed Smart Contract
QLoRA Finetuning
Contract 1 : Fix 1
Contract 2 : Fix 2
...
Contract 3 : Fix 3
Contract-Fix paired Dataset

Fig. 1. Architecture of the system

3) QLoRA (Quantized Low-Rank Adaptation): For efficient fine-tuning, we employ QLoRA, a parameterefficient fine-tuning technique from the Peft (ParameterEfficient Fine-Tuning) library. QLoRA extends the concept of Low-Rank Adaptation (LoRA) by incorporating quantization, further reducing the memory footprint and computational requirements during fine-tuning. This approach allows us to fine-tune large language models while minimizing the number of trainable parameters.

B. Experimental Setup and Evaluation

The research experiment was performed on an NVIDIA A100 40GB GPU. The CodeLlama model was quantized to 4 bits using QLoRA and fine-tuned using PEFT for 3 epochs, with a batch size of 4, and using cross-entropy loss.

C. Data Preparation

The proposed architecture employs a two-step process—Vulnerability Detection and Vulnerability Remediation—utilizing a pre-trained model and finetuning techniques to address security threats effectively. Crucially, before these steps can be executed, it is essential to convert the data into the appropriate format through data preprocessing and cleaning. This foundational process ensures that the data fed into large language models (LLMs) is accurate, consistent, and ready for reliable analysis, thereby enhancing the overall effectiveness of the security framework.

1) About Dataset: The dataset used for detection of vulnerabilities in smart contracts is the Slither-AuditedSmart-Contracts dataset from Hugging Face. It is a multilabeled dataset to determine whether the smart contract is safe or detect the presence of the following vulnerabilities- Access Control, Arithmetic, Reentrancy, Unchecked Calls, Locked Ether, Bad Randomness, Double Spending. The AutoMESC dataset uses seven of the most well-known smart contract security tools to label contracts based on vulnerability. It also includes the corresponding fix for the vulnerable smart contracts which forms the paired dataset.

2) Conversion to Instruction Dataset: In this work, the data must to converted to the Instruction Dataset format prior in order to train the LLM. The flow diagram of dataset creation is described in 2.

Preprocessing is an essential step in preparing raw data and code for large language models, as they cannot directly interpret unprocessed input. This process involves cleaning, organizing, and transforming the data to ensure it is suitable for model consumption. Customized regular expressions are employed to remove noise such as punctuation marks, numerals, special characters, and unnecessary blank spaces from the dataset. Following this, the cleaned data is tokenized, breaking it down into smaller units or tokens that represent meaningful elements, such as keywords or identifiers in code. This ensures the data is structured, consistent, and in a format that the model can effectively learn from, ultimately enhancing the model's ability to detect and remediate vulnerabilities.

A prompt template and instruction dataset for training large language models (LLMs) serve as crucial components in the development and fine-tuning of these models. The prompt template outlines the structure and format for generating prompts to elicit specific responses from the model. It typically includes placeholders for variables or prompts tailored to the desired task. Meanwhile, the

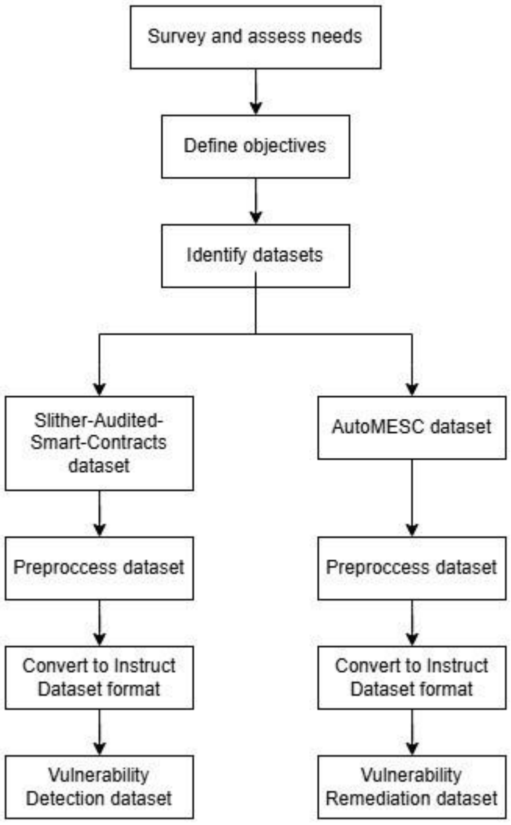

Survey and assess needs
Define objectives
Identify datasets
Slither-Audited-Smart-Contracts dataset
AutoMESC dataset
Preproccess dataset
Preproccess dataset
Convert to Instruct Dataset format
Convert to Instruct Dataset format
Vulnerability Detection dataset
Vulnerability Remediation dataset

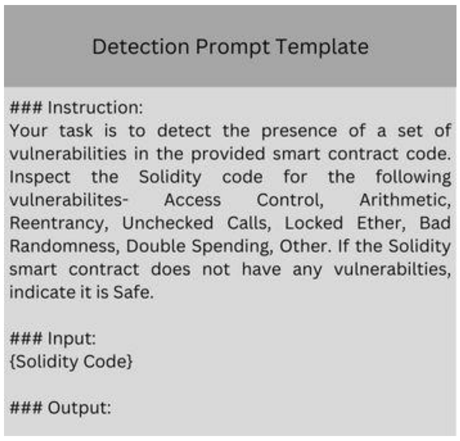
Fig. 2. Flow diagram of Instruction Dataset creation

instruction dataset consists of examples paired with specific instructions or directives for the model to follow. Fig. 3 shows the prompt templates used to create the instruct dataset used for fine tuning the model on the task of vulnerability detection. The Detection Prompt Template defines the instruction and input provided to the LLM. The output is a multilabel classification of the vulnerabilities detected. Similarly, 4 shows the prompt templates used to create the instruct dataset used for fine tuning the model on the task of vulnerability remediation. The Fixing Prompt Template defines the instruction and input provided. The output returned is the smart contract devoid of vulnerabilities. By exposing the model to a wide range of examples with clear instructions, it learns to produce more accurate and contextually appropriate responses.

D. Model and Training

The architecture is based on a two-step process: Vulnerability Detection and Vulnerability Remediation. These steps utilize a pre-trained model and fine-tuning techniques to effectively address security threats.

Model used for the finetuning tasks: The model chosen for the task is CodeLlama-7b-Instruct-Solidity. This is a Large Language Model with 7 billion parameters. It has been finetuned on over 6000 smart contracts to generate Solidity smart contracts.



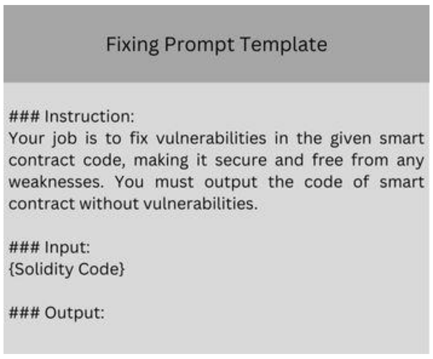
Detection Prompt Template

Instruction:
Your task is to detect the presence of a set of vulnerabilities in the provided smart contract code. Inspect the Solidity code for the following vulnerabilites- Access Control, Arithmetic, Reentrancy, Unchecked Calls, Locked Ether, Bad Randomness, Double Spending, Other. If the Solidity smart contract does not have any vulnerabilties, indicate it is Safe.

Input:
{Solidity Code}

Output:

Fig. 3. Prompt Template for Vulnerability Detection



Fixing Prompt Template

Instruction:
Your job is to fix vulnerabilities in the given smart contract code, making it secure and free from any weaknesses. You must output the code of smart contract without vulnerabilities.

Input:
{Solidity Code}

Output:

Fig. 4. Prompt Template for Vulnerability Remediation

- PEFT: PEFT, short for Parameter-Efficient FineTuning, stands as a solution aimed at adapting large pretrained models to diverse downstream applications in a more cost-effective manner. Traditional methods of fine-tuning require adjusting all parameters of a model, which can be excessively resource-intensive. However, PEFT adopts a different approach by fine-tuning only a select few additional model parameters. This strategy drastically reduces computational and storage demands while maintaining performance levels comparable to fully fine-tuned models. By minimizing the number of parameters altered, PEFT makes it more feasible to train and deploy large language models (LLMs) on consumer-grade hardware. Integrated seamlessly with libraries like Transformers, Diffusers, and Accelerate, PEFT streamlines the process of loading, training, and utilizing large models for inference, thereby offering a faster and more accessible pathway to leveraging powerful language models in various applications.

- QLoRa: There are various ways of achieving Parameter efficient fine-tuning. Low-Rank Adaptation LoRA ,QLoRA are the most widely used and effective.QLoRA stands out as an enhanced version of LoRA, offering greater memory efficiency. Going beyond the advancements of LoRA, QLoRA introduces quantization of the weights of LoRA adapters to even lower precision, such as 4-bit instead of the previous 8-bit. This refinement significantly diminishes the memory footprint and storage requirements. When employing QLoRA, the pre-trained model is loaded into GPU memory with weights quantized to 4-bit precision, in contrast to the 8-bit precision utilized in LoRA. Despite this reduction in bit precision, QLoRA maintains a comparable level of effectiveness to its predecessor, demonstrating its ability to optimize memory usage without compromising performance. This the Codellama model was loaded in a quantized manner and finetuned using QLoRA.

- Supervised Fine-tuning: Supervised fine-tuning of large language models (LLMs) is a crucial step in adapting these pre-trained models to specific downstream tasks. In this process, the LLM, which has been previously trained on a large corpus of text data, is fine-tuned on a smaller, task-specific dataset annotated with labels.

-

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

IV. RESULTS

1) Evaluation Metric and Performance Analysis: To assess the effectiveness, efficiency, and accuracy of our models in detecting and mitigating vulnerabilities in smart contracts, we utilize the pass@k metric. This metric helps evaluate how often the correct result is within the top k predictions made by the model. By analyzing pass@k, we can measure the reliability of the system, compare different techniques, and identify the most effective approaches. This evaluation method provides a robust framework for analyzing and improving vulnerability detection and mitigation in smart contracts.pass@k measures the probability that the correct solution appears in the top k attempts as shown in the following equation.

$$\text{pass @k} = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}$$

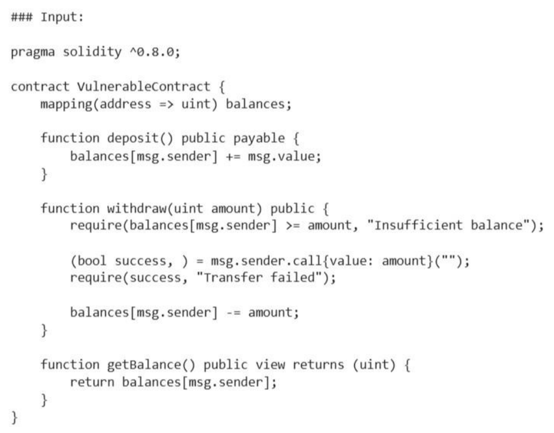
where n is the number of generated samples, c is the number of correct outputs, and k is the number of top-ranked outputs considered. The pass@k metric is a valuable tool for evaluating large language models (LLMs) by measuring their ability to generate correct outputs within the top k attempts. Specifically, it assesses how often a model's correct response appears among the top k predictions when multiple outputs are generated for a given prompt. This metric is particularly useful in scenarios where the model may produce several plausible answers, and it is important to know if the correct answer is among the most likely ones. Pass@k provides insights into the model's performance across different tasks, helping to gauge its reliability, versatility, and potential areas for improvement. By analyzing this metric, researchers and developers can better understand the strengths and limitations of an LLM, leading to more targeted refinements and enhancements.The exploit mitigation rate (EMR) is calculated as:

$$\text{EMR} = \frac{\text{Exploits prevented}}{\text{Total vulnerabilities tested}} \times 100\%$$

This quantifies how effective the patching mechanism is in neutralizing real-world attack vectors.

2) Experimental Data set: To support the research and development efforts of the project, open-source datasets are utilized to create a vulnerability database consisting of smart contracts with known vulnerabilities. This database serves as a critical resource for testing and refining detection and mitigation techniques. Specifically, a set of 14 contracts, each containing documented vulnerabilities, was selected and employed to evaluate the performance of the detection and mitigation tasks.

One such sample contract, in which a known vulnerability is present, namely re-entrancy. This vulnerability occurs when a contract allows an external call to another untrusted contract before it resolves its internal state changes, such as updating balances. Take an example smart contract with known vulnerability in 5.

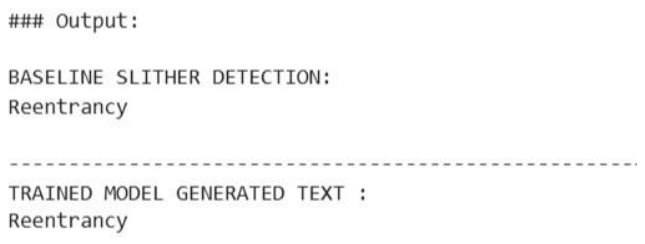


```
### Input:

pragma solidity ^0.8.0;

contract VulnerableContract {
    mapping(address => uint) balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw(uint amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");

        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");

        balances[msg.sender] -= amount;
    }

    function getBalance() public view returns (uint) {
        return balances[msg.sender];
    }
}
```

Fig.5.ContractwithReentrancyVulnerability

This allows a malicious contract to repeatedly call the withdraw function before the balance is deducted, effectively draining all funds from the contract. 6 shows the vulnerability detector successfully identifies the reentrancy issue in the contract, flagging the external call before the balance update as a critical security risk. This detection highlights the need to reorder operations to prevent potential exploits.

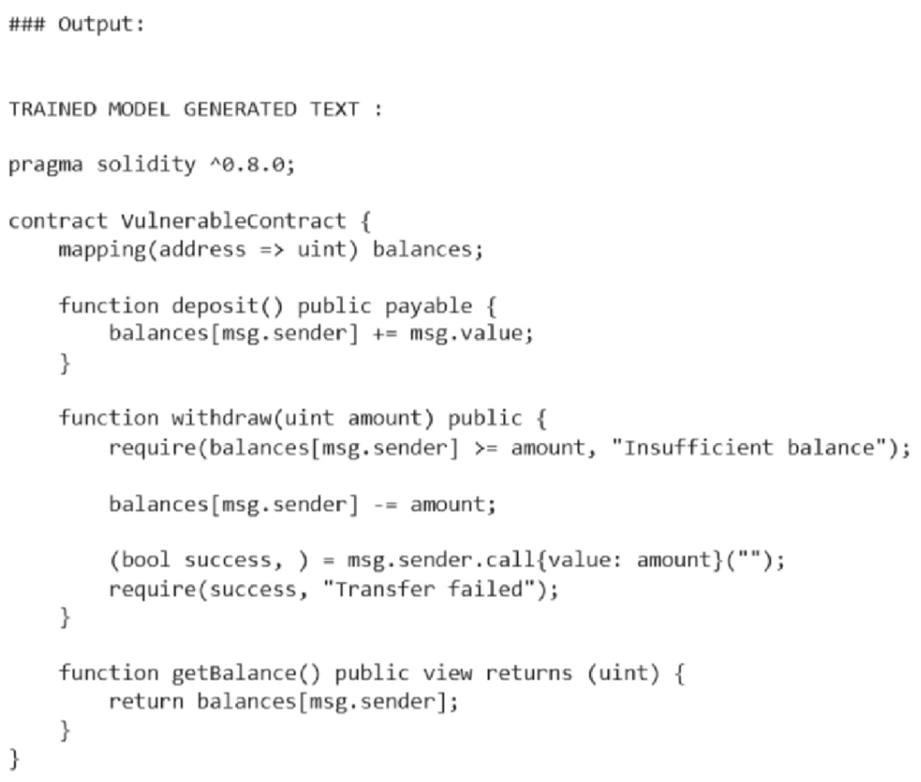
The vulnerability mitigator addresses the reentrancy vulnerability in 7 by reordering the operations within the withdraw function. Specifically, it modifies the contract to update the sender's balance before making the external call that transfers Ether. In the original vulnerable code, the Ether transfer was performed first, leaving the contract open to reentrancy attacks where a malicious contract could re-enter the withdraw function multiple times before the balance was updated. By moving the balance deduction to



```
### Output:

BASELINE SLITHER DETECTION:
Reentrancy

----------------------------------------------------
TRAINED MODEL GENERATED TEXT :
Reentrancy
```

Fig. 6. Detection of Vulnerability

occur immediately after the balance check and before the external call, the mitigator ensures that even if a reentrant call is made, the attacker's balance has already been reduced, preventing them from exploiting the contract to withdraw more funds than they are entitled to. This change effectively eliminates the reentrancy vulnerability.



```
### Output:


TRAINED MODEL GENERATED TEXT :

pragma solidity ^0.8.0;

contract VulnerableContract {
    mapping(address => uint) balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw(uint amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");

        balances[msg.sender] -= amount;

        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
    }

    function getBalance() public view returns (uint) {
        return balances[msg.sender];
    }
}
```

Fig. 7. Mitigation of Vulnerability

The results in TABLE III and figure 8 highlight the performance of the models for both vulnerability detection and mitigation tasks, measured using the pass@k metric. For vulnerability detection, the model demonstrates solid performance with a pass@1 of 48.2%, meaning that the correct vulnerability is identified almost half the time on the first attempt. As more predictions are considered, the model's effectiveness increases, with pass@5 reaching 53.6% and pass@10 at 57.4%, indicating a consistent improvement when multiple predictions are allowed.

TABLE III. PASS@K METRICS FOR VULNERABILITY DETECTION AND MITIGATION

Task	pass@1	pass@5	pass@10
Vulnerability Detection	48.2%	53.6%	57.4%
Vulnerability Mitigation	42.3%	46.5%	51.3%

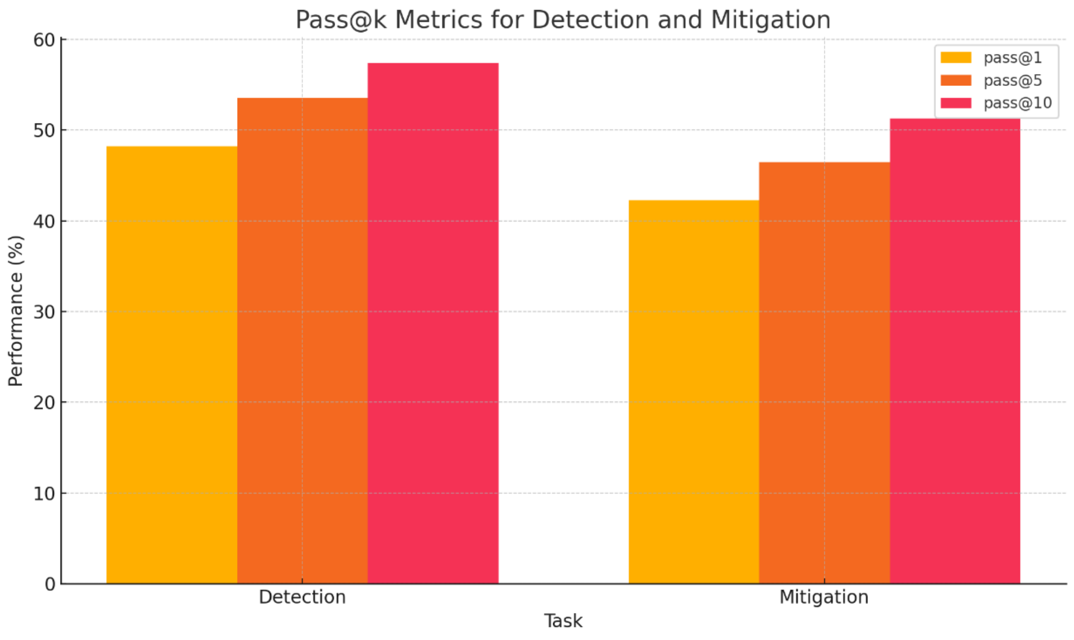


Figure 8. Model's performance using the pass@k metric.

In the vulnerability mitigation task, the model also shows strong results, with a pass@1 of 42.3%. This means that in over 42% of cases, the correct mitigation is identified in the first prediction. The performance continues to improve with higher values of k, as evidenced by a pass@5 of 46.5% and pass@10 of 51.3%. These metrics reflect the model's capability to generate effective solutions, with a notable increase in accuracy as additional suggestions are considered.

The measured performance scores—pass@1 of 48.2% for detection and 42.3% for mitigation, reaching up to pass@10 scores above 50%—can be explained by a number of architectural and procedural choices within our approach. To begin with, employing CodeLlama-7b-Instruct-Solidity as a model specific to programming and following instructions offered a sound starting point for understanding domain-specific language. Second, our fine-tuning approach with QLoRA and precision-efficient training enabled us to successfully convert the base model on a handpicked dataset of Solidity vulnerabilities, picking up patterns such as reentrancy and unchecked calls. Additionally, through prompt template construction that separated out vulnerabilities and necessitated explicit reasoning on fix strategies, we made it possible for the model to acquire structured vulnerability-to-fix mappings. This correspondence between prompt format, training data, and domain specificity greatly enhanced the chances that the correct output would be in the top-k predictions. Finally, the small but well-chosen set of 14 actual contracts helped provide cleaner learning signals, which, although perhaps constraining generalization, facilitated better targeted performance in this test.

The comparative analysis between leading smart contract security tools in the table IV, emphasizes the unique strengths and weaknesses of existing tools with our model. SmartCheck, as rule-based, has moderate precision but low recall (<30%) and thus a low F1-score (~0.25). This reflects its poor capacity to generalize outside of pre-defined patterns, frequently failing to detect important vulnerabilities. Mythril is marginally better in accuracy with fewer false positives but still has low recall (~27%), constraining its overall F1-score (~0.40). ContractFix is solely concerned with fixing, using external tools for vulnerability detection. Although it effectively fixes ~84% of identified vulnerabilities, it has an estimated exploit mitigation rate of only ~35%, which is indicative of its reliance on the quality of previous detection and the inflexibility of its template-based fix logic. Conversely, our LLM-based method sidesteps such limitations by housing both detection and remediation in a unified adaptive architecture. Although standard precision-recall metrics were not directly calculated because of the generation-based output of LLM, our system posts a high pass@5 score of around 53.56% and a pass@10 of 57.4%, showing consistent generation of correct patches in the top choices. Most importantly, it achieves an exploit mitigation rate of 51.3 %—considerably better than ContractFix—displaying its ability to generate safe and functionally equivalent repairs. This places our model as an integrated, data-driven solution addressing detection coverage as well as patch efficacy in practice.

TABLE IV: TOOL COMPARISION METRICS

Tool	Precision	Recall	F1-score	Pass@k	Exploit Mitigation
SmartCheck[23]	Moderate (rule-based)	Low (<30%)	Low (~0.25)	N/A (no repair)	N/A
Mythril[13]	High (few false alarms)mdpi.com	Low (~27%)	Low (~0.40)	N/A (no repair)	N/A
ContractFix[28]	– (uses other tools for detection)	–	–	N/A (no repair)	~84% of found vulns (≈35% exploits)
Our Model	-	-	-	High (pass@10 ≈57.4%)	High(pass@10 ≈ 51.3%), improver over higher k.

V. CONCLUSION AND FUTURE ENHANCEMENTS

The proposed architecture introduces a dynamic and innovative approach to enhancing Ethereum smart contract security by integrating advanced machine learning techniques with Generative AI and Large Language Models (LLMs). This two-step process—Vulnerability Detection followed by Vulnerability Remediation—not only identifies security threats but also generates context-aware patches to mitigate these vulnerabilities. The fine-tuning of the LLM on a dataset of smart contracts and their associated vulnerabilities allows the model to discern between vulnerable and fortified contracts effectively, ensuring a more robust detection and remediation process.

To the best of our knowledge, this is one of the first approaches to leverage LLMs for the automatic repair of vulnerabilities in smart contracts, marking a significant milestone in the field. In an era of increasing reliance on Generative AI, such a methodology is crucial for addressing the growing complexity and scale of security threats in the blockchain ecosystem. The approach's adaptability, combined with the potential for continuous improvement through the inclusion of high-quality datasets, ensures that the system remains responsive to emerging vulnerabilities. This ability to evolve alongside the landscape of security threats offers a powerful tool for developers and stakeholders, significantly reducing the risk of exploitation and ensuring the integrity of smart contracts in a rapidly advancing digital age. Future enhancements consist of enlarging the dataset with real world bug reports from Code4rena and improving multi-vulnerability patching with ensembles of prompts, in addition to implementing an automated verification loop using SMT solvers.

REFERENCES

[1] Li Zhang, Xie Yongping, Yang Zheng, Wei Xue, Xianrong Zheng, and Xiaobo Xu. The challenges and countermeasures of blockchain in finance and economics. *Systems Research and Behavioral Science*, 37, 06 2020.

[2] Philip Treleaven, Richard Gendal Brown, and Danny Yang. Blockchain technology in finance. *Computer*, 50(9):14–17, 2017.

[3] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[4] G. Kannan, Manjula Pattnaik, G. Karthikeyan, Balamurugan E., P. John Augustine, and Lohith J J. Managing the supply chain for the crops directed from agricultural fields using blockchains. *2022 International Conference on Electronics and Renewable Systems (ICEARS)*, pages 908–913, 2022.

[5] Abhishek H Bharath Kumar Lohith J. J Nikhil C, Hetav Pabari. A review on supply chain management in agriculture empowered by ethereum and ipfs. *International Conference on Computational Intelligence and Digital Technologies*, 2021.

[6] Sudarshan Parthasarathy, Akash Harikrishnan, Gautam Narayanan, Lohith J, and Kunwar Singh. Secure distributed medical record storage using blockchain and emergency sharing using multi-party computation. pages 1–5, 04 2021.

[7] Randeep Singh, Bilal Ahmed Mir, Lohith J J, Dhruva Sreenivasa Chakravarthi, Adel R Alharbi, Harish Kumar, and Simon Karanja Hingaa. Smart healthcare system with light-weighted blockchain system and deep learning techniques. *Computational intelligence and neuroscience*, 2022:1621258, 2022.

[8] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack. *Journal of Cases on Information Technology (JCIT)*, 21(1):19–32, 2019.

[9] Nick Szabo. Formalizing and securing relationships on public networks. *First monday*, 1997.

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

[11] Stefano Bistarelli, Gianmarco Mazzante, Matteo Micheletti, Leonardo Mostarda, Davide Sestili, and Francesco Tiezzi. Ethereum smart contracts: Analysis and statistics of their source code and opcodes. *Internet of Things*, 11:100198, 2020.

[12] Shafaq Khan, Faiza Loukil, Chirine Ghedira, Elhadj Benkhelifa, and Anoud Bani-Hani. Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-Peer Networking and Applications*, 14, 09 2021.

[13] Hanting Chu, Pengcheng Zhang, Hai Dong, Yan Xiao, Shunhui Ji, and Wenrui Li. A survey on smart contract vulnerabilities: Data sources, detection and repair. *Information and Software Technology*, page 107221, 2023.

[14] Cipai Xing, Zhuorong Chen, Lexin Chen, Xiaojie Guo, Zibin Zheng, and Jin Li. A new scheme of vulnerability analysis in smart contract with machine learning. *Wireless Networks*, pages 1–10, 2020.

[15] Pooja Srinivasan et al. Tp-detect: trigram-pixel based vulnerability detection for ethereum smart contracts. *Multimedia Tools and Applications*, pages 1–15, 2023.

[16] Yuhang Sun and Lize Gu. Attention-based machine learning model for smart contract vulnerability detection. In *Journal of physics:conference series*, volume 1820, page 012004. IOP Publishing, 2021.

[17] Oliver Lutz, Huili Chen, Hossein Fereidooni, Christoph Sendner, Alexandra Dmitrienko, Ahmad Reza Sadeghi, and Farinaz Koushanfar. Escort: ethereum smart contracts vulnerability detection using deep neural network and transfer learning. *arXiv preprint arXiv:2103.12607*, 2021.

[18] Weichu Deng, Huanchun Wei, Teng Huang, Cong Cao, Yun Peng, and Xuan Hu. Smart contract vulnerability detection based on deep learning and multimodal decision fusion. *Sensors*, 23(16):7246, 2023.

[19] Mojtaba Eshghie, Cyrille Artho, and Dilian Gurov. Dynamic vulnerability detection on smart contracts using machine learning. In *Evaluation and assessment in software engineering*, pages 305–312. 2021.

[20] Nami Ashizawa, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. Eth2vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In *Proceedings of the 3rd ACM international symposium on blockchain and secure critical infrastructure*, pages 47–59, 2021.

[21] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, 8(2):1133–1144, 2020.

[22] TonTon Hsien-De Huang. Hunting the ethereum smart contract: Color-inspired inspection of potential attacks. *arXiv preprint arXiv:1807.01868*, 2018.

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

[24] Lohith J, Kunwar Singh, and Bharatesh Chakravarthi. Digital forensic framework for smart contract vulnerabilities using ensemble models. *Multimedia Tools and Applications*, pages 1–44, 11 2023.

[25] Hai Jin, Zeli Wang, Ming Wen, Weiqi Dai, Yu Zhu, and Deqing Zou. Aroc: An automatic repair framework for on-chain smart contracts. *IEEE Transactions on Software Engineering*, 48(11):4611– 4629, 2022.

[26] Xiao Liang Yu, Omar I. Al-Bataineh, David Lo, and Abhik Roychoudhury. Smart contract repair. *CoRR*, abs/1912.05823, 2019.

[27] Tai D. Nguyen, Long H. Pham, and Jun Sun. sguard: Towards fixing vulnerable smart contracts automatically. *CoRR*, abs/2101.01917, 2021.

[28] Pengcheng, Peng, Yun, Qingzhao, Tao, Dawn, Prateek, Sanjeev, Zhuotao, and Xusheng. Contractfix: A framework for automatically fixing vulnerabilities in smart contracts, 2023.

[29] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Evmpatch: Timely and automated patching of ethereum smart contracts, 2020.

[30] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. Smartshield: Automatic smart contract protection made easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 23–34, 2020.

[31] Abdulaziz Alhefdhi, Hoa Khanh Dam, Thanh Le-Cong, Bach Le, and Aditya Ghose. Adversarial patch generation for automated program repair, 2023.

[32] Harshit Joshi, Jose Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek,´ and Gust Verbruggen. Repair is nearly generation: Multilingual program repair with llms, 2022.

[33] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms, 2023.

[34] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Practical program repair in the era of large pre-trained language models, 2022.