Identifying and Fixing Vulnerable Patterns in Ethereum Smart Contracts: A Comparative Study of Fine-tuning and Prompt Engineering Using Large Language Models

Marco Ortu[b] (marco.ortu@unica.it), Giacomo Ibba[a] (giacomo.ibba@unica.it), Claudio Conversano[b] (conversa@unica.it), Roberto Tonelli[a] (roberto.tonelli@unica.it) , Giuseppe Destefanis[c] (giuseppe.destefanis@brunel.ac.uk)

[a] Dept. of Computer Science and Mathematics, University Of Cagliari, Via Porcell 4, Cagliari (IT)

[b] Dept. of Business and Economics Sciences, University Of Cagliari, Viale Fra Ignazio 17, Cagliari (IT)

[c] Dept. of Computer Sciences, Brunel University, London (UK)

Corresponding Author:
Marco Ortu
University Of Cagliari, Italy
Tel: (+39) 070-675-3316
Email: marco.ortu@unica.it

Identifying and Fixing Vulnerable Patterns in Ethereum Smart Contracts: A Comparative Study of Fine-tuning and Prompt Engineering Using Large Language Models

Marco Ortu[b,*], Giacomo Ibba[a], Claudio Conversano[b], Roberto Tonelli[a], Giuseppe Destefanis[c]

[a]*Dept. of Computer Science and Mathematics, University Of Cagliari, Via Porcell 4, Cagliari (IT)*
[b]*Dept. of Business School, University Of Cagliari, Viale Fra Ignazio 17, Cagliari (IT)*
[c]*Dept. of Computer Sciences, Brunel University, London (UK)*

Abstract

We investigate the use of GPT-3.5's ChatGPT and Google's Bard for automated program repair (APR) in Solidity smart contracts. Three distinct experimental setups were examined to evaluate these models as APR tools. Our analysis found that ChatGPT consistently performed better than Google's Bard across all setups. While Bard successfully addressed some vulnerabilities, it exhibited certain limitations in its capacity for automated smart contract repair.

Our results indicate that the most effective methodology involves providing the models with the vulnerable code snippet, listing all the exposed lines of code, and specifying the associated vulnerabilities. This approach achieved an accuracy rate of 89%, successfully repairing the majority of smart contracts. Thus, it appears that ChatGPT and Bard can be effectively utilized as APR tools for smart contracts when used in tandem with specific vulnerability detection applications.

Our study also identified limitations related to the tools' ability to analyze large contracts. This issue could be partially mitigated by segmenting larger contracts into smaller parts for analysis, but ideally, the tools should be able to scan entire contracts in one operation. This approach requires an in-depth understanding of the contract and a

*Corresponding author.
Email addresses: `marco.ortu@unica.it` (Marco Ortu), `giacomo.ibba@unica.it` (Giacomo Ibba), `conversa@unica.it` (Claudio Conversano), `roberto.tonelli@unica.it` (Roberto Tonelli), `giuseppe.destefanis@brunel.ac.uk` (Giuseppe Destefanis)

comprehensive evaluation using multiple vulnerability detection tools to ensure robust coverage of potential vulnerabilities. Despite these constraints, our research highlights the potential of using AI language models in automated smart contract repair.

Keywords: Automatic Program Repair, Natural Language Processing, ChatGPT, Solidity, Smart Contract, Smart Contract Vulnerabilities

1. Introduction

In 2015, Ethereum brought about the innovation of smart contracts, computer protocols that facilitate, verify, and enforce the negotiation or performance of a digital contract. These protocols were created with the intent to facilitate secure transactions between two parties, bypassing the need for a trusted third-party authority. The initial programs were tailored towards simple functions such as facilitating interactions with the blockchain, managing token distribution, and performing basic operations related to currency distribution, collection, and management.

As the Solidity programming language evolved, it enabled the development of more complex applications, such as Initial Coin Offering programs Ibba et al. (2018), Crowdsales, Role Playing Games, and programs handling ERC-20 Tokens Chen et al. (2020) and Non-Fungible-Tokens (NFTs) Wang et al. (2021). Furthermore, SCs form the foundation of decentralized applications (DApps), specifically decentralized financial applications (DeFi) Jensen et al. (2021), which offer a range of services including lending, borrowing, and asset management. Like all software, smart contracts can be affected by vulnerabilities Singh et al. (2020). Exploitations in financial applications could lead to irreversible financial losses and severe consequences Bracamonte & Okada (2017). Some of the most critical vulnerabilities that can affect a Solidity program Atzei et al. (2017) include:

- Reentrancy: This vulnerability occurs when attackers exploit the functionality of smart contracts and their interactions with external programs, enabling them to execute unauthorized code and reenter the program.
- Denial of Service (DoS): DoS attacks render smart contracts ineffective either

temporarily or permanently by employing various techniques, such as externally manipulated infinite loops or restricting specific operations to the contract's owner.

- Arithmetic Overflows and Underflows: Overflows and underflows can occur when fixed-size variables try to store numeric values or data that exceed their capacity.

- Unchecked Low-Level Calls: This vulnerability is caused by incorrect use of the call() function, particularly when the validation of its return value is not adequately conducted.

- Time Dependency: Contracts often use timestamps within critical functions related to the transfer of funds. However, the 'block.timestamp' variable can be manipulated by a miner, indicating the need for careful usage.

- Bad Randomness: Due to inherent blockchain properties, achieving true randomness in Solidity is challenging. Developers often resort to pseudo-random generation as a workaround.

- Access Control: This vulnerability pertains to the unauthorized access to structs, variables, and functions that should ideally be accessible only to specific users.

- Front Running: This issue arises from the mining process. The time taken to mine transactions allows an attacker to send a transaction and include it in a block before the original transaction.

- Short Address: A Short Address Attack occurs when an attacker manipulates the data sent in an Ethereum transaction, misleading the smart contract into reading more data than was sent.

In response to these vulnerabilities and their potential consequences, developers have developed diagnostic and detection tools. Some of these tools identify vulnerabilities and highlight the affected code lines without suggesting fixes. Others provide comprehensive reports that include guidance on how to remediate the identified issues.

However, none of the existing tools offer automated solutions for repairing problematic Solidity programs, which could greatly improve developer efficiency.

This work explores new methods for Automated Program Repair (APR) Goues et al. (2019) of smart contracts using Natural Language Processing (NLP) Chowdhary & Chowdhary (2020) techniques. Specifically, it investigates the potential use of OpenAI's [1] models, such as ChatGPT, to automatically repair Solidity programs and compares this to Google's Bard [2]. While both ChatGPT and Codex models have demonstrated effectiveness in correcting Python, Java, and Javascript programs, their performance in automated repair of SCs is yet to be explored. This work considers three methods for using ChatGPT and Bard as APR tools:

- The direct input of SCs, with the models expected to identify and rectify potential vulnerabilities. Here, ChatGPT and Bard serve as vulnerability detection and APR tools.
- The input of a test subset of SCs, including known vulnerabilities and their fixes. This approach involves training ChatGPT and Bard using supervised learning and then testing them with another subset of SCs.
- The input of SCs that include known vulnerabilities and the corresponding affected code lines. In this case, ChatGPT and Bard serve only as APR tools, with vulnerabilities identified by dedicated tools.

The structure of this paper is as follows: Section 2 provides the background and reviews related works. The methodologies employed are described in Section 3. Section 4 presents the results, while the subsequent analysis is detailed in Section 5. A discussion of these results can be found in Section 6, followed by an exploration of the study's limitations in Section 7. Future research directions are suggested in Section 8, and finally, Section 9 concludes the paper.

[1] https://openai.com/

[2] https://bard.google.com/

2. Background And Related Works

2.1. Vulnerability Detection Tools and APR on Smart Contracts

Various tools and techniques have been developed to detect vulnerabilities within Solidity smart contracts. For instance, Reguard Liu et al. (2018) uses a two-step method. It first converts the smart contract code into an intermediate representation, then transforms this intermediate representation to C++. The outcome is a bug report which includes all potential reentrancy patterns. Osiris Osi (2018) is another tool specifically designed to find integer bugs in Ethereum smart contracts. Slither Feist et al. (2019) employs Static Single Assignment (SSA) form along with a minimal instruction set to maintain the semantic information that might be lost during the transformation from Solidity to bytecode.

Mythril Di Angelo & Salzer (2019) is a security analysis tool for EVM bytecode, aimed at identifying vulnerabilities in smart contracts across various EVM-compatible blockchains. Mythril combines symbolic execution, SMT solving, and taint analysis to detect a wide array of security vulnerabilities.

Contemporary Automatic Program Repair (APR) methods generally fall into two categories: program synthesis Durieux & Monperrus (2016) and program transformation Visser (2001). Program synthesis methods utilize genetic algorithms Huang et al. (2010), constraint solving, and machine learning techniques Gupta et al. (2017) to create new, correct code that fixes bugs. On the other hand, program transformation methods use techniques such as program analysis, abstract interpretation Logozzo & Martel (2013), and formal verification Nguyen et al. (2019) to modify the original program in a way that resolves the bug.

Different APR techniques have been proposed specifically for smart contract systems. Some of these techniques focus on identifying and patching known vulnerabilities such as reentrancy, integer overflow, and denial-of-service attacks, while others aim to repair more general bugs caused by logic errors. One example is SmartShield Zhang et al. (2020), a tool that takes a smart contract as input and produces secure EVM bytecode free of the following insecure patterns:

- State changes following external calls.

- Absence of checks for out-of-bound arithmetic operations.
- Absence of checks for failing external calls.

Another approach involves using a genetic programming search algorithm Yu et al. (2020) to patch vulnerable smart contracts. The tool sGUARD Nguyen et al. (2021) compiles a smart contract into a JSON file that contains the bytecode, source map, and abstract syntax tree (AST). The bytecode is utilized to detect vulnerabilities, while the source map and AST assist in repairing the smart contract at the source code level. To our knowledge, only one study so far has used the GPT model for automatically patching buggy programs. This study was conducted on JavaScript programs and fine-tuned the GPT-2 model to generate source code. The aim was to train the model with as many code snippets as possible, thereby enabling GPT to learn the context of the provided programming language. To the best of our knowledge, our proposed approach would be the first to use GPT models for delivering APR techniques that specifically target smart contract vulnerability patching.

2.2. *Current Literature limitations*

Research on Ethereum Smart Contract vulnerability detection highlights several gaps in the field. Many of these gaps stem from the inherent nature of smart contracts. These contracts are often written in newer programming languages such as Solidity, which can be difficult to analyse and can present security vulnerabilities due to possible interactions with external contract functions or interfaces. Once deployed, smart contracts are unchangeable, preventing the patching of potential security issues (Rameder et al., 2022).

Existing methods for vulnerability detection face several shortcomings. Many depend on fixed rules set by experts, leading to challenges with scalability, high false alarm rates, and a single detection type. These methods are of limited use and can be time-consuming. Some newer techniques utilize machine learning and deep learning for automated vulnerability detection, but these methods can only provide binary detection for smart contracts and suffer from poor interpretability, particularly when dealing with smart contracts which contain multiple vulnerabilities (Qian et al., 2022).

Despite an increase in research in this area over the past three years, these challenges highlight the need for more sophisticated, scalable, and precise detection methods. Future research should aim to address these limitations to enhance the security and reliability of Ethereum smart contracts.

2.3. Statement of Purpose

This study aims to contribute to the field of Ethereum Smart Contract (SC) vulnerability detection and repair, a critical task given the sensitive and diverse applications of SCs. While numerous detection tools exist to help developers identify vulnerabilities, addressing them remains a complex task. The wide variety of SC applications, from currency management to tracking food supply and managing healthcare systems, demands an efficient and automated approach for repairing vulnerabilities, reducing potential risks such as financial loss and data theft. We aim to examine the applicability of GPT-3 and Bard models in the Automated Program Repair of SCs, focusing specifically on the comparison between model fine-tuning and prompt engineering. We intend to test the performance of different models, including davinci, ada, babbage, curie, ChatGPT, and the improved codex model, gpt 3.5 turbo and Bard. The ultimate goal of this research is to enhance the accuracy and efficiency of Automated SCs Repair, improving the overall security and reliability of Ethereum smart contracts.

2.4. Practical Implications

The methodology proposed in this study could benefit practitioners in several ways. First, from a security perspective, the methods in this study could improve the security of Ethereum smart contracts. The automated identification and repair of vulnerabilities could reduce the chance of financial loss or other severe consequences. This is essential for practitioners who depend on smart contracts for various operations, such as decentralized finance applications.

From an efficiency and cost-saving point of view, the automatic repair of vulnerabilities could save significant time and resources for developers. Currently, fixing issues is a manual process that can be labor-intensive and expensive. If an AI model can be trained to repair these issues automatically, it would free up developers' time to focus on other aspects of smart contract development, potentially increasing productivity.

From a trust perspective, addressing and repairing these vulnerabilities could potentially increase overall confidence in blockchain technology as studies have shown that development practices can have a significant impact on cryptocurrency prices Bartolucci et al. (2020). If smart contracts become more secure and reliable, more businesses and individuals may be motivated to use blockchain technologies in their operations. This could increase the variety and scale of opportunities available for practitioners in the blockchain sector.

3. Research Methodology

Figure 1 shows our study and experimental design. There are three main phases: Ethereum Smart Contracts vulnerability detection, Language Models fine-tuning and fix generation.

3.1. ChatGPT

ChatGPT has been applied in a variety of domains and use cases. It's been used to assist individuals with health decisions by offering insights based on data Biswas (2023b). Similarly, in the context of environmental studies, ChatGPT has been employed to support climate change research Biswas (2023a), providing data-backed insights and helping in the interpretation of complex research. Within academia, its use has extended to librarianship Lund & Wang (2023), providing assistance in information retrieval and organization. In terms of Automatic Program Repair (APR), studies have examined ChatGPT's utility in program debugging and bug resolution Destefanis et al. (2023); Surameery & Shakor (2023), highlighting its potential and limitations. Yet, its application in addressing vulnerabilities in Solidity smart contracts is still uncharted territory, largely due to unique aspects of Solidity that differentiate it from other programming languages.

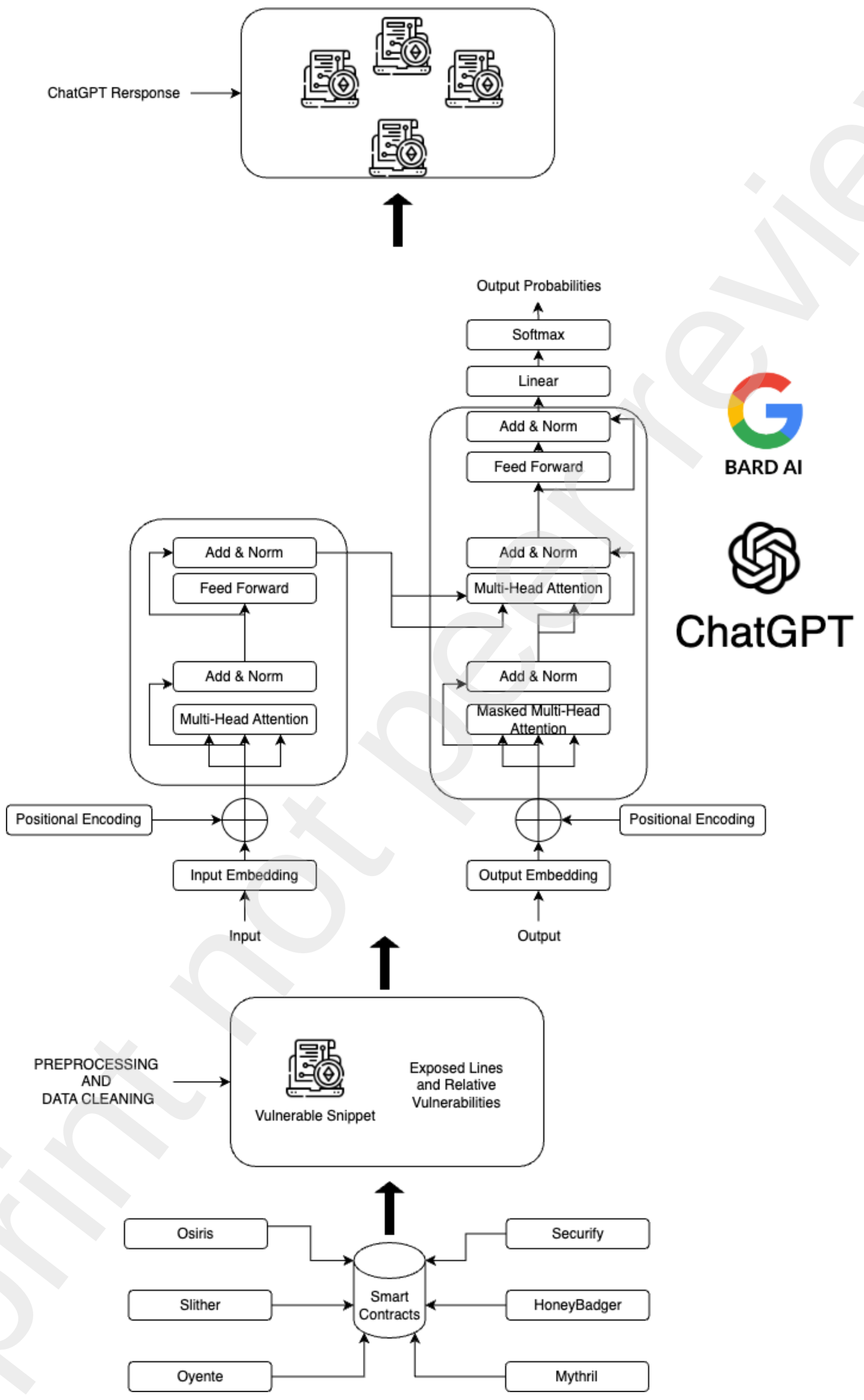


Figure 1: The proposed research methodology

3.2. Bard

Bard, an AI model with natural language generation capabilities, has shown good performance in a variety of tasks related to language understanding and generation. Its architecture and training data are not publicly available, yet its outputs reveal an ability to produce contextually relevant text. Bard has been recognized for producing creative output, demonstrating an ability to generate text that fits well within given contexts. Bard, like GPT-3.5, does not have a conscious understanding of the content it produces. Rather, it identifies patterns in its training data and generates outputs based on these patterns. Further exploration could reveal the extent of its applicability in different contexts, including potential use in Automatic Program Repair (APR).

3.3. Codex Models and Chat Models

The Codex models have been trained on both natural language and code of several programming languages. OpenAI suggests that Codex can be used for tasks like code completion, adding comments to code, turning comments into code, and finding specific APIs for applications. According to the current literature, Codex has impressive performance in code completion, outclassing other open-source models across all parameters. OpenAI's Codex models have been tested for zero-shot vulnerability repair, and it has been proven that by giving a suitable repair prompt, these models can outperform the others, also considering that Codex models have been previously trained on a tremendous amount of data. Another work proving that Codex is very effective (despite not being explicitly designed for APR) relies on testing the models on a benchmark of buggy algorithms implementations, comparing them to existing neural APR approaches, and proving to be highly competitive. Currently, Codex is deprecated and replaced by *GPT 3.5 turbo*, a chat model capable of executing the same tasks as Codex models. To our knowledge, this model has not been tested for Automated-Program-Repair. Still, considering that it replaces Codex, we may think of it as a direct evolution and a better version. Therefore, we may also consider the existing work already done for APR with Codex as part of the GPT 3.5 turbo as well.

3.4. Dataset

In this study, we used a dataset from our previous research to evaluate ChatGPT and BARD performances in the context of smart contract Automatic Program Repair (APR) Ibba et al.. Our dataset combines SmartCorpus Pierro et al. (2020), an organized repository of smart contracts, and SmartBugs Ferreira et al. (2020), which contains 47k contracts, including a subset of pre-labeled vulnerable contracts. One of the objectives of this research is to examine ChatGPT and BARD capabilities as diagnostic tools for detecting smart contract vulnerabilities. However, it is essential to scan the subset of smart contracts that ChatGPT and BARD will analyze. To identify vulnerabilities in this subgroup of smart contracts, we used the following tools:

- HoneyBadger.
- Osiris.
- Oyente.
- Mythril.
- Slither.
- Securify.

Osiris is specifically designed to detect arithmetic bugs, reentrancy, improper uses of the call function, and time manipulation bugs Ibba & Ortu (2022). HoneyBadger focuses on identifying honeypots within smart contracts, while Oyente targets a limited subset of vulnerabilities. Slither provides a comprehensive suite of vulnerability detection capabilities and offers suggestions for code optimization, enhanced contract comprehension, and code review assistance. Mythril, known for its high accuracy in detecting vulnerabilities, analyzes smart contracts developed for EVM-compatible blockchains; however, it requires more time to execute and analyze even smaller programs due to its relatively slower performance. In this study, we extracted vulnerable code snippets identified by the diagnostic tools for each contract. The first three tools were exclusively used to scan contracts with a pragma version less than or equal to

0.4.26, as they only support those versions. These tools were able to detect additional vulnerabilities worth including in our analysis. The other three detection tools were employed to scan all programs, irrespective of the pragma version.

To speed up the process of verifying ChatGPT and BARD responses, we manually wrote patches for each contract. It is important to stress that the pragma version is particularly relevant to our analysis, as different versions may require distinct patch patterns.

3.5. Patching Process

We used a supervised learning strategy to train the model. This process involved supplying the model with examples of vulnerable code snippets alongside their corresponding fixes for a selection of each vulnerability type. To ensure the effectiveness of the provided fixes, we conducted re-analysis of the smart contracts using three diagnostic tools. If these tools did not identify any vulnerabilities, we considered the fix to be correct.

Different types of vulnerabilities require specific mitigation methods. For instance, there are several methods to mitigate reentrancy. One method uses the checks, effects, and interactions (CEI) pattern. Another method uses a mutex to secure the function call that could be susceptible to reentrancy. A third method applies a pull payment mechanism that employs an escrow as an intermediary for fund transfers, thereby preventing direct interactions with potentially harmful contracts. In our study, we decided to use the mutex method to prevent reentrancy.

The mutex method employs a boolean flag, also known as a ”lock”, to secure the function call that might be prone to reentrancy vulnerability. Initially, the lock is set to the ”unlocked” state (represented by false). Before the execution of the potentially vulnerable function begins, the lock is switched to the ”locked” state (represented by true). After the function execution is concluded, the lock is reverted back to the ”unlocked” state (represented by false).

```
function withdrawAll() public {
        uint oCredit = credit[msg.sender];
        if (oCredit > 0) {
            balance -= oCredit;
            bool callResult = msg.sender.call.value(oCredit)();
            require (callResult);
            credit[msg.sender] = 0;
        }
    }
}
```

Listing 1: Example of withdraw function vulnerable to a reentrancy attack

A malicious actor might manipulate the call function by constructing harmful code. This would allow them to re-enter an arbitrary point within the withdraw function, possibly leading to the draining of Ethereum. In the following, the introduction of a reentrancy guard is explored as a modification to the code aimed at mitigating this vulnerability.

```

bool private locked = false;

function withdrawAll() public {
        uint oCredit = credit[msg.sender];
        require(!locked);
        locked = true;
        if (oCredit > 0) {

            balance -= oCredit;
            bool callResult = msg.sender.call.value(oCredit)();
            require (callResult);
            credit[msg.sender] = 0;
        }
    }
    locked = false;
}
```

Listing 2: Reentrancy attack prevented using a mutex guard

The code remains susceptible to reentrancy, but the inclusion of the 'locked' variable in the 'require' statement effectively counters exploitation attempts by reversing the transaction. Consequently, an unauthorized user is denied the opportunity to reenter the withdraw function code.

Denial of Service (DoS) attacks on Solidity Smart Contracts can present in a variety of forms, necessitating diverse patching approaches. One such instance concerns the inappropriate execution of operations exclusive to the owner, such as the selfdestruct function. Consider a situation where a contract, holding a substantial Ether amount, experiences a misfired selfdestruct call. This leads to the contract becoming unusable and the remaining ETH being permanently locked within the Smart Contract. Another instance involves denial of access to essential contract functions. An attacker might add a significant number of participants to a funds-distributing contract, exceeding gas consumption costs and making payment unfeasible. As a result, the contract becomes incapable of fund distribution, essentially rendering it useless. To reduce DoS attack risks, it is imperative to guard against external manipulations of mappings or arrays. A demonstration of Denial of Service might be the following:

```
function refundAll() public {
    for(uint x; x < refundAddresses.length; x++) { // arbitrary
        length iteration based on how many addresses participated
    // <yes> <report> DENIAL_OF_SERVICE
        require(refundAddresses[x].send(refunds[refundAddresses[x]])
            ); // doubly bad, now a single failure on send will hold
             up all funds
    }
}

}
```

Listing 3: Example of Denial of Service while refunding participants

The previous example demonstrates a refunding mechanism in which all participants receive refunds simultaneously. In this scenarios involving a substantial number of participants, a single failure in the send function can impede the release of funds for all individuals. To address this concern, a straightforward solution entails incorporating a payment logic that enables individual refunds for participants, as outlined below:

```
function refundAll() public {
    for(uint x; x < refundAddresses.length; x++) {
        require(refund(refundAddresses[x]));
    }
}

    function refund(address recipient) private returns (bool) {
        require(refunds[recipient] > 0);
        uint amount = refunds[recipient];
        refunds[recipient] = 0;
        return recipient.send(amount);
    }
```

Listing 4: Example of Denial of Service Fix

Before transferring funds to the recipient, we ensure the refund amount exceeds zero. Moreover, we set the refund amount to zero before executing the transfer to prevent potential reentrancy attacks. This way, even if the transfer triggers a fallback function that attempts to call 'refundAll' again, it won't cause an endless loop of transfers.

Arithmetic vulnerabilities, which usually result from storing values outside the range of a variable's data type during program execution, are relatively straightforward to address. A common mitigation strategy is the use of the SafeMath library, which provides arithmetic functions designed to prevent overflows and underflows. Developers are often encouraged to implement this library as a means of avoiding arithmetic vulnerabilities. However, another possible strategy to address arithmetic vulnerabilities implies taking advantage of require statements and modifiers, preventing user

```
pragma solidity ^0.4.15;
library SafeMath {
  function add(uint256 a, uint256 b) internal pure returns (uint256
      c) {
    c = a + b;
    assert(c >= a);
    return c;
  }

  function sub(uint256 a, uint256 b) internal pure returns (uint256
      c) {
    assert(b<=a);
    c = a - b;
    return c;
  }
  function mul(uint256 a, uint256 b) internal pure returns (uint256
      c) {
  if (a == 0) {
  return 0;
  }
  c = a * b;
  assert(c / a == b);
  return c;
  }

  function div(uint256 a, uint8 b) internal pure returns (uint256) {
  assert(b > 0);
  uint256 c = a / b;
  assert(a == b * c + a % b);
  }


}

 contract Overflow {

     using SafeMath for uint;

     uint private sellerBalance=0;

     function add(uint value) returns (bool){

         sellerBalance = sellerBalance.add(value);


     }

 }
```

Listing 5: Example of SafeMath implementation to prevent arithmetic overflows

Additional vulnerabilities in contracts may include unprotected calls to functions that send Ether to arbitrary addresses.

```
function BalancedPonzi() {
    currentNiceGuy = msg.sender;
    beta = msg.sender;
}
function funnel() {
    beta.send(this.balance);
}
```

Listing 6: Example of function sending ETH to an arbitrary address

This vulnerability can be readily addressed by restricting functions that transfer Ether to only send currency to authorized addresses. This can be achieved by implementing a mapping that stores all authorized users and a procedure that allows the contract's owner to approve specific addresses. Notably, the owner is automatically granted authorization at the time of contract creation.

```
mapping(address=>bool) private authorized;

function BalancedPonzi() {

    currentNiceGuy = msg.sender;
    authorized[currentNiceGuy] = true;
    beta = msg.sender;
    authorized[beta] = true;
}

function authorize(address x)
{
    require(msg.sender == currentNiceGuy);
    authorized[x] = true;
}

function funnel() {
    require(authorized[beta] == true);
    beta.send(this.balance);
}
```

Listing 7: The function now allows to send ETH only to authorized addresses

A user could directly assign a fixed value to the array length, making all the storage accessible. Consequently, an attacker could potentially access the entire array storage and assign arbitrary values to each cell. To mitigate this vulnerability, users should add values incrementally as needed or prevent user-controlled variables from determining the array length assignment. Furthermore, the array should be limited to the maximum available storage slots (2^{256}-1).

The Unchecked Low-Level Calls (ULLC) vulnerability arises when the success of a call function is not adequately evaluated, potentially affecting subsequent operations that depend on it. To address this issue, it is crucial to thoroughly assess the success of the call function and incorporate a 'require' statement that verifies the function's effectiveness before executing any subsequent operations that rely on its results.

```
function Command(address adr,bytes data)
payable
public
{
    require(msg.sender == Owner);
    bool success = adr.call.value(msg.value)(data);
    require(success);
}
```

Listing 8: Example of how to handle the Low-Level Call function

The Short Address attack implies an attacker exploiting a vulnerability in the data transmitted within an Ethereum transaction. By intentionally providing truncated data, the attacker aims to deceive the smart contract into interpreting a shorter address length than what was actually transmitted. This manipulation can potentially lead to unintended behavior or security vulnerabilities within the smart contract. For instance, suppose we have the following code snippet:

```
 // <yes> <report> SHORT_ADDRESSES
 function sendCoin(address to, uint amount) returns(bool
     sufficient) {
     if (balances[msg.sender] < amount) return false;
     balances[msg.sender] -= amount;
     balances[to] += amount;
     Transfer(msg.sender, to, amount);
```

```
        return true;
    }
```

Listing 9: Example of Short Addresses attack vulnerable snippet

The Short address attack could be exploited in this particular function to increase the amount sent to the address *'to'*. To fix this vulnerability, the code could be patched as outlined below:

```
function sendCoin(address to, uint amount) returns(bool sufficient)
    {
        // Check for a valid length of the 'to' address
        require(to != address(0));
        require(msg.data.length >= (2 * 32) + 4); // 2 * 32 bytes
            for the 'to' address + 4 bytes for the 'amount'

        if (balances[msg.sender] < amount) return false;
        balances[msg.sender] -= amount;
        balances[to] += amount;
        Transfer(msg.sender, to, amount);
        return true;
    }
```

Listing 10: Patch for Short Addresses attacck

The fixed code ensures that the *'to'* address does not match with the '0' address, such as to avoid sending tokens to an invalid address. Moreover, we check that the length of the transaction's data is at list *'(2 * 32) + 4'*, which is the expected size for the *to* and *amount* parameters.

The Front-Running attack consists of re-ordering the transactions pool such that an attacker can gain advantages. Usually, transactions with higher prices are mined first; consequentially, an attacker can send a transaction with a higher gas price such that the attacker's transaction is mined first.

```
pragma solidity ^0.4.16;

contract EthTxOrderDependenceMinimal {
    address public owner;
    bool public claimed;
    uint public reward;

    function EthTxOrderDependenceMinimal() public {
        owner = msg.sender;
    }

    function setReward() public payable {
        require (!claimed);

        require(msg.sender == owner);
        // <yes> <report> FRONT_RUNNING
        owner.transfer(reward);
        reward = msg.value;
    }

    function claimReward(uint256 submission) {
        require (!claimed);
        require(submission < 10);
        // <yes> <report> FRONT_RUNNING
        msg.sender.transfer(reward);
        claimed = true;
    }
}
```

Listing 11: Example of Front Running vulnerable SC

Such as other SCs vulnerabilities, the Front-Running attack can be mitigated with different approaches. However, the commit-reveal scheme is among the best solutions. This scheme represents a cryptographic algorithm that enables an individual to commit to a value while maintaining its confidentiality from others yet retaining the ability to disclose it later. The values in a commitment scheme possess a binding property, ensuring they remain unalterable once committed. The scheme encompasses two distinct phases: a commit phase, during which a value is selected and specified, and a reveal phase, in which the value is unveiled and subjected to verification. The previous code patched using a commit-reveal scheme is outlined below:

```
pragma solidity ^0.4.16;

```

```
contract EthTxOrderDependenceMinimal {
    address public owner;
    bool public claimed;
    uint public reward;
    mapping(address => bytes32) public commitments;

    event RewardSet(address indexed owner, uint reward);
    event RewardClaimed(address indexed claimer, uint reward);

    constructor() public {
        owner = msg.sender;
    }

    function setReward(bytes32 commitment) public payable {
        require(!claimed);
        require(msg.sender == owner);
        require(commitments[msg.sender] == 0);

        reward = msg.value;
        commitments[msg.sender] = commitment;

        emit RewardSet(msg.sender, reward);
    }

    function revealReward(uint256 submission) public {
        require(!claimed);
        require(commitments[msg.sender] != 0);
        require(keccak256(abi.encodePacked(submission)) ==
            commitments[msg.sender]);

        delete commitments[msg.sender];
        owner.transfer(reward);

        emit RewardClaimed(msg.sender, reward);
        claimed = true;
    }
}
```

Listing 12: Patched Front-Running exposed SC using the commit-reveal scheme

In this patched contract, the *setReward()* function takes a commitment as input, which is a hash of the actual reward value. Then, the owner sets the reward by providing the commitment and sending the corresponding Ether to the contract. Later, the owner reveals the actual reward value by calling the *revealReward()* function, which checks that the submitted value matches the stored commitment and transfers the reward to the owner.

Access Control exposures concern vulnerable patterns where unauthorized users access functions, structs, and variables that should be bounded with access constraints. This vulnerability could lead to dire consequences, such as unauthorized ETH withdrawal, inoperability of contracts, etc. An example of an access control vulnerable snippet is outlined below:

```
pragma solidity ^0.4.22;

contract Phishable {
   address public owner;

   constructor (address _owner) {
       owner = _owner;
   }

   function () public payable {} // collect ether

   function withdrawAll(address _recipient) public {
       // <yes> <report> ACCESS_CONTROL
       require(tx.origin == owner);
       _recipient.transfer(this.balance);
   }
}
```

Listing 13: Example of Access Control vulnerable code snippet

In this code snippet, only the contract owner should be authorized to withdraw funds. However, the logic relies on *tx.origin* to check if the caller is the contract's owner, which is extremely unsafe. Indeed *tx.origin*, represents the original sender of the transaction, and is not necessarily the caller of the function. Therefore, this specific vulnerable snippet could be patched by switching *tx.origin* with *msg.sender*, which instead represents the immediate caller of the function, and it is the best logic to check if the caller is the contract's owner.

```
pragma solidity ^0.4.22;

contract Phishable {
    address public owner;

    constructor(address _owner) {
        owner = _owner;
    }

    function () public payable {} // collect ether

    function withdrawAll(address _recipient) public {
        require(msg.sender == owner);
        _recipient.transfer(address(this).balance);
    }
}
```

Listing 14: Fixed vulnerable Access Control Snippet

3.6. *Prompt Engineering Process*

In this section, we discuss the Prompt Engineering Process and explore several ways ChatGPT can be used to automatically repair smart contracts.

The first method involves providing ChatGPT with both the buggy contract and the corresponding patched version. After supplying the model with a substantial sample of examples covering various vulnerabilities, we can evaluate its performance by asking it to fix a vulnerable contract without providing the patch. A potential drawback of this approach is that ChatGPT might not completely adhere to the manually provided guidelines and instead combine them with its own understanding of smart contract vulnerability fixes, potentially resulting in a non-compilable program.

The second method utilizes ChatGPT both as a diagnostic tool to identify vulnerabilities and as an Automatic Program Repair tool. Here, ChatGPT is given only the vulnerable contract and is asked to detect and fix vulnerabilities based on its knowledge of Solidity smart contracts. While this approach grants complete control to ChatGPT, it may not be the optimal solution. It assumes that ChatGPT can identify and fix all vulnerabilities within a smart contract, which might only be feasible for simple examples and not for more complex cases. Additionally, this strategy could be risky due to potential pragma version conflicts, which could result in non-compilable programs.

The final method combines ChatGPT with ad-hoc diagnostic tools specifically designed for smart contract vulnerability detection. The first phase involves running tools to scan a substantial sample, annotating the vulnerable lines with the corresponding vulnerabilities. The second phase provides ChatGPT with the vulnerable contract and the annotations derived from the diagnostic process. Notably, we indicate the vulnerable code line and the associated vulnerability, explicitly asking ChatGPT to fix the contract while maintaining the given pragma version, except in cases where version updates are necessary. This approach enables ChatGPT to understand what it needs to improve, and since the vulnerability is explicitly specified, it should also know how to fix the problematic line properly. Moreover, enforcing the given pragma version reduces the likelihood of receiving a non-compilable program.

4. Results

In this section, we present the results of our experiments. Before discussing the outcomes, it is essential to clarify the criteria for determining if the output program generated by the two tools is considered correct. A returned smart contract (SC) by ChatGPT and Bard is deemed valid if all reported vulnerabilities have been fixed and the contract still compiles. The response of both tools is considered incorrect under any of the following conditions:

- The exposed code lines remain uncorrected (even a single vulnerability left unaddressed is considered a failure).
- A partial fix is deemed a failure. For instance, if ChatGPT handles an arithmetic overflow or underflow by using SafeMath functions but does not import the SafeMath library, it is considered a partial fix.

- The contract does not compile.
- The pragma version is altered.

We observed that, unless explicitly instructed otherwise, both tools tend to switch the pragma version to the most recent one while retaining deprecated elements incompatible with the latest Solidity version. Consequently, we explicitly requested the tools to fix the contract without modifying the pragma version.

It should be emphasized that the exposures related to a Solidity file are considered patched only if this file is compilable. If the examined Solidity file does not compile, then all the vulnerabilities are considered un-fixed. On the other hand, if the contract compiles, we check the single exposed LOCs, and those with a proper fix are deemed successful. The methodology divides the Solidity files into:

- Non-Compilable (NC): After patching, the source code contains errors that make the file un-compilable.
- Non-Analyzable (NA): The tool does not scan and analyze the smart contract. Therefore, the vulnerabilities are considered unpatched.
- Successful (S): The Solidity file is scanned and appropriately analyzed. The tool's fixes allow the contract to compile successfully.

We collected a sample of 199 vulnerable SCs. However, we could provide as input the complete source code for the majority of the Solidity files, but for those too big in terms of lines of code (LOC), we had to provide only the functions that included the exposures. ChatGPT and Bard will successfully handle programs not exceeding 200 LOC. Our dataset of vulnerable SCs provides the following vulnerabilities:

- Short Address: One instance is susceptible to the short address attack.
- Arithmetic overflows and underflows: We have 105 LOC exposed to arithmetic overflows and underflows.
- Access Control: 18 LOC are vulnerable to access control.
- Bad Randomness: Bad randomness affects 8 LOC.

- Denial Of Services: 7 LOC could lead to denial of service.
- Front Running: 4 instances are exposed to front-running attacks.
- Reentrancy: In our dataset, 90 possible reentrant patterns are included.
- Time Manipulation: 5 examples that could lead to attacks due to time manipulation.
- Unchecked Low-Level Calls: 56 lines of code could lead to attacks due to ULLC.

4.1. Preliminary General Comparison Between ChatGPT and Bard

This subsection makes a preliminary general comparison between the two AIs based on our observations of the behavior of both ChatGPT and Bard during the experiment. First, we must consider that ChatGPT and Bard rely on different models. ChatGPT is based on the Generative Pre-Trained Transformer, while Bard is based on the Language Model for Dialogue Applications (laMDA). Therefore, Bard's input and output are based on dialogues, and ChatGPT takes advantage of natural language to interpret and generate text. Moreover, Bard is connected to the internet, while GPT 3.5 is not. Indeed, when fixing a vulnerability or honeypot, Bard will cite the source where it took the knowledge to patch the smart contract. In contrast, ChatGPT takes advantage of its prior knowledge given by the massive training on a large body of text from various sources. Concerning the practical aspect, we observed that sometimes Bard would not even scan the SC, even if we considered a program with a few lines of code. At the request of fixing the program, Bard will return only a statement like 'I'm a language model, and I'm not designed for this task.' At first, we thought the program was too big to be analyzed by Bard, but we observed that the same problem occurred even with more minor smart contracts. Therefore, we can not state precisely why Bard does not even scan a specific smart contract, but we can only make assumptions:

- The first assumption implies that Bard misunderstands the request and mistakes the task for another it cannot perform.
- For some reason, Bard does not have temporary access to the needed sources for the SC repair.

- The problem could be due to Bard being at his experimental version and some programming bugs.

Moreover, we observed that in contrast with ChatGPT, Bard does not consider the suggestions given by the prompt in case of a wrong fix. For instance, assume that Bard patches an arithmetic overflow/underflow vulnerable SC using the SafeMath library functions but without importing the library itself. Suppose the user tries to train Bard, suggesting, for example, explicitly importing the SafeMath library when dealing with arithmetic vulnerabilities. In that case, Bard still will not import it, in contrast with ChatGPT.

4.2. ChatGPT and Bard as an APR Tool with Exposed Lines and Associated Vulnerabilities

The experimental setup incorporates ChatGPT and Bard as Automated Program Repair (APR) tools employed in conjunction with vulnerability detection tools. Remarkably, we ran the vulnerability detection tools described in previous sections, scanning all our samples. In this session, we give as input a prompt p that can be described as follows: *' Try to fix this Solidity smart contract with the given exposed lines of code. You will be provided with the source code and the specific vulnerable snippet. Remember that you are not allowed to switch the pragma version.'* The source code and the exposed LOCs with the associated vulnerability follow the request. For instance, the LOCs are given as follows:

- the line 'overflow_example += 200;' is exposed to arithmetic overflow.
- the line ' bool res = msg.sender.call.value(amount)();' is exposed to a reentrancy attack.
- the line 'lastInvestmentTimestamp = block.timestamp;' is exposed to time manipulation.

Therefore, this methodology consists of supervising the APR process, taking advantage of the prior knowledge given by the detection tool scanning.

The performances of ChatGPT as an automatic program repair tool provided with the exposed lines returned an accuracy of almost 95%. It follows that the majority of our smart contracts were adequately patched. Remarkably, looking at the table, only five smart contracts were not compilable, and three were non-analyzable. Therefore we

Category	NC	NA	S
Short Address	0	0	1
Overflow and Underflow	0	0	16
Front Running	0	0	4
Reentrancy	1	0	30
Denial of Service	0	0	6
Unchecked Low-Level Calls	1	3	47
Bad Randomness	0	0	8
Time Manipulation	0	0	5
Access Control	0	0	18
Multi-Vulnerable	3	2	55

Table 1: Number of Non-Compilable, Non-Analyzable and Successful Smart Contracts Samples in the ChatGPT APR Process

can state that overall, ChatGPT can generate syntactically correct patches for vulnerable smart contracts. Remarkably, for the non-compilable contracts, the syntactically wrong patches were:

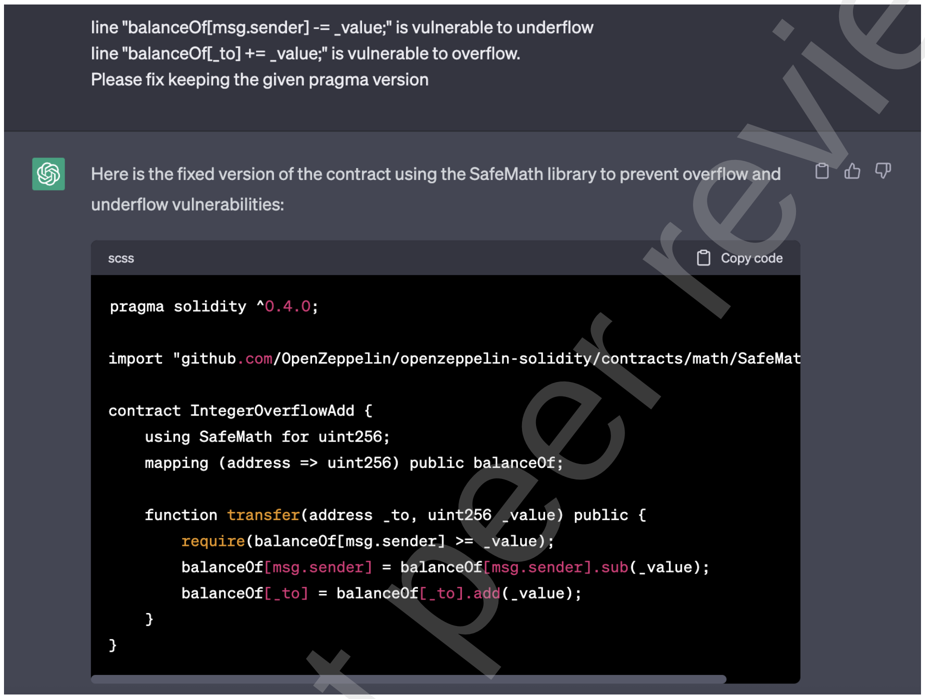
- Bad check of the success of the function call in the ULLC contracts: after the call function exposed line ChatGPT put a require with the call function that returned a compilation error.
- Wrong use of mutex guards in the Reentrancy exposed contracts: ChatGPT wrote a mutex guard pattern without declaring the boolean variable.
- ChatGPT twice imported the SafeMath library from ’openzeppelin’ in contracts under 0.4 pragma in the Multi-Vulnerable contracts. The error of the last non-

compilable contract was due to the use of features not included in the pragma version of the analyzed program.

Category	Correct ChatGPT	Incorrect ChatGPT
Short Address	1	0
Overflow and Underflow	97	8
Front Running	4	0
Reentrancy	72	18
Denial of Service	6	1
Unchecked Low-Level Calls	54	2
Bad Randomness	15	0
Time Manipulation	5	0
Access Control	24	0

Table 2: Number of fixed exposures by ChatGPT

Looking at Table 2 highlighting the results of the ChatGPT analysis, we observe that the most significant number of wrongly patched contracts concerns those exposed by reentrancy. Remarkably, sometimes ChatGPT does not add any additional pattern or best practice to prevent reentrant functions. We assume that this problem is correlated to the implementation of the 'send' and 'transfer' functions; indeed, all the wrong patches include the use of these two functions. We hypothesize ChatGPT considers 'send' and 'transfer' generally safe enough to prevent reentrancy since they bound the execution with gas limit constraints. Nevertheless, they do not provide complete protection against reentrancy; an additional patch is required. Another small subset of SCs exposed to reentrancy includes misusing mutex guards. Mainly, ChatGPT used a boolean lock to prevent reentrant patterns, but without declaring it. Most other exposures were adequately patched, except for a few vulnerable arithmetic instances. The main problem relies upon using the SafeMath library made by ChatGPT. Remarkably, ChatGPT imported SafeMath from GitHub, which led to a compilation error. The other issue is that ChatGPT uses the SafeMath functions without importing the library. Although the small subset of bad-fixed SCs, we can state that ChatGPT can successfully patch Solidity programs by taking advantage of the methodology described in this section.


line "balanceOf[msg.sender] -= _value;" is vulnerable to underflow
line "balanceOf[_to] += _value;" is vulnerable to overflow.
Please fix keeping the given pragma version

Here is the fixed version of the contract using the SafeMath library to prevent overflow and underflow vulnerabilities:

```
pragma solidity ^0.4.0;

import "github.com/OpenZeppelin/openzeppelin-solidity/contracts/math/SafeMat

contract IntegerOverflowAdd {
    using SafeMath for uint256;
    mapping (address => uint256) public balanceOf;

    function transfer(address _to, uint256 _value) public {
        require(balanceOf[msg.sender] >= _value);
        balanceOf[msg.sender] = balanceOf[msg.sender].sub(_value);
        balanceOf[_to] = balanceOf[_to].add(_value);
    }
}
```

Figure 2: Example of the formulated request to ChatGPT

The capability of Bard to analyze a SC could be considered as much at the same level as ChatGPT in terms of LOC number and contract complexity. The performances of Bard regarding contract scanning and analysis are resumed in Table 1.

Table 3 highlights that the number of NC and NA contracts is significantly higher in the Bard analysis. Regarding NA contracts, we do not know precisely the criteria for which a Solidity file is considered analyzable by Bard. In the initial stage, we thought this problem could be due to the excessive number of LOCs, but we found out that Bard did not analyze even contracts with just 30 LOCs several times. Therefore, the criteria for whether Bard analyzes a contract or not remain esoteric. It should be emphasized that of 31 known files vulnerable to Reentrancy, none has been appropriately patched; indeed, we have 0 S samples. Before explaining why this happens, further

Category	NC	NA	S
Short Address	1	0	0
Overflow and Underflow	10	1	5
Front Running	0	0	4
Reentrancy	26	5	0
Denial of Service	1	2	3
Unchecked Low-Level Calls	13	13	25
Bad Randomness	3	4	1
Time Manipulation	1	2	2
Access Control	3	5	10
Multi-Vulnerable	14	4	42

Table 3: Number of Non-Compilable, Non-Analyzable and Successful Smart Contracts Samples in the Bard APR Process

elucidations about Reentrancy are needed. A user may use the call, send, and transfer functions for ETH transfer. Generally, the call function is considered insecure for ETH transfer. In contrast, the transfer and send functions bound the execution with gas limits and therefore is considered safe concerning the call function. Nonetheless, send and transfer are still regarded as insecure if they are not appropriately managed with mutex, modifiers, and check and effects patterns. Therefore, despite being generally secure, send and transfer should not be considered reliable to avoid Reentrancy. Remarkably, Bard incorrectly patched all the samples that used the call function to send ETH. To be more precise, Bard tried to fix this re-entrant pattern by adding a try-catch and, finally constructs (this last one is not available in Solidity, and it follows that the file will not compile). On the other hand, most re-entrant patterns due to the send and transfer functions are correctly patched. Concerning arithmetic overflows and underflows, Bard fixes these vulnerabilities using two methodologies. The first one consists of explicitly adding modifiers and require statements to check that the arithmetic data does not exceed the allowed size for the specific variable. The other methodology relies upon the

Category	Correct Bard	Incorrect Bard
Short Address	0	1
Overflow and Underflow	28	77
Front Running	4	0
Reentrancy	29	61
Denial of Service	3	4
Unchecked Low-Level Calls	21	35
Bad Randomness	3	12
Time Manipulation	2	3
Access Control	14	10

Table 4: Number of fixed exposures by Bard

use of the SafeMath library. However, Bard mainly uses this library's functions without importing it, which leads to a compilation error and unfixed vulnerable patterns. For this very reason, we conducted a further session for experimental purposes. When Bard gave us the response, including the smart contract using the SafeMath functions, but without importing the library, we tried to ask him to fix the following Solidity file by explicitly importing the SafeMath. However, after a few attempts, we observed that Bard did not follow the instructions, and it kept making the same error. We tried the same with the 'call' re-entrant pattern, where we asked Bard not to use the try-catch-finally paradigm, but we asked instead to use boolean locks to protect the code snippet. However, also in this case, Bard did not follow our instructions. We want to highlight that comments significantly impact the analysis' quality. Indeed, we observed that (especially when dealing with Bard) the SC samples with high-quality comments are the most successfully patched.

4.3. *ChatGPT as a Vulnerability Detection and APR Tool*

This methodology provides ChatGPT and Bard as vulnerability detection and automatic program repair tools. In this experiment session, we also want to test the tools' ability to spot and fix the vulnerabilities without any suggestions inside our sample.

Category	ChatGPT's Accuracy	Bard's Accuracy
Short Address	100%	0%
Overflow and Underflow	92%	27%
Front Running	100%	100%
Reentrancy	80%	32%
Denial of Service	86%	43%
Unchecked Low-Level Calls	96%	37%
Bad Randomness	100%	20%
Time Manipulation	100%	40%
Access Control	100%	58%

Table 5: First Experimental Setup Accuracy for Single Vulnerability.

Trivially, to deliver this experiment, we must delete all the comments of our dataset referring to the exposed LOCs. Since we already have proven that accurate comments improve the quality of the analysis, not deleting them would completely destroy this experiment. Therefore, we removed all the comments referring to vulnerabilities or the exposed lines of code. The prompt *p* to give to both tools could be built as follows: *'Can you spot and fix the vulnerabilities inside this Solidity smart contract? Remember that you are not allowed to switch the pragma version.'* After this statement, the tool should be provided with the source code without the original or at least those comments referring to exposures. In this experiment session, we included a further rule which defines whether or not the success of the analysis. Indeed, since we are taking advantage of ChatGPT and Bard as vulnerability detection tools, the rule states that if a vulnerability is not detected by the tool, that specific exposure is considered unpatched. As for the experiment session where ChatGPT was used as an APR tool, providing the vulnerable lines, most of the smart contracts were returned as compilable. The main problem in this session relies on the fact that ChatGPT successfully detects the principal vulnerabilities (arithmetic, reentrancy, access control). Still, as highlighted in Table 7, it does not adequately spot time manipulation, front running,

Category	NC	NA	S
Short Address	0	0	1
Overflow and Underflow	3	0	13
Front Running	0	0	4
Reentrancy	2	0	29
Denial of Service	1	0	5
Unchecked Low-Level Calls	0	2	49
Bad Randomness	0	1	7
Time Manipulation	0	0	5
Access Control	0	0	18
Multi-Vulnerable	7	4	49

Table 6: Number of Non-Compilable, Non-Analyzable and Successful Smart Contracts Samples in the ChatGPT Vulnerabilities Detection and APR Process

Category	Correct ChatGPT	Incorrect ChatGPT
Short Address	0	1
Overflow and Underflow	89	16
Front Running	0	4
Reentrancy	66	24
Denial of Service	5	2
Unchecked Low-Level Calls	41	15
Bad Randomness	8	7
Time Manipulation	0	5
Access Control	20	4

Table 7: Number of detected and fixed exposures by ChatGPT

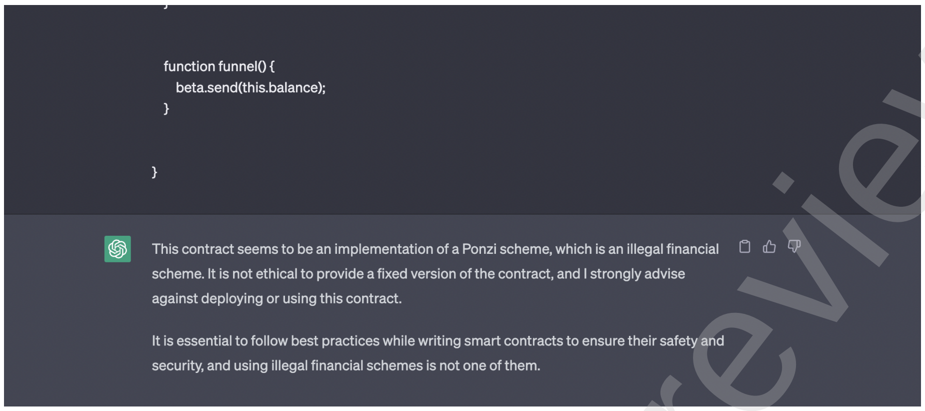
and a few bad randomness exposures. A particularly exciting sample of our dataset was not analyzed because ChatGPT identified it as a scam contract. Remarkably, this contract implements a Ponzi scheme 3, which is a fraudulent investment scheme that operates by promising high returns to early investors using funds contributed by later investors. ChatGPT refused to scan this contract, stating that it would be unethical to scan and help us detect vulnerabilities inside this program. Moreover, besides detecting and fixing the exposed LOCs, ChatGPT advises how to implement best practices and how some code patterns should be changed to improve the source code's security.

In contrast with ChatGPT, Bard returned several non-compilable programs besides those not analyzed regardless after the scanning, vulnerability detection, and fixing processes. However, in contrast with ChatGPT, Bard successfully detected a subset of vulnerabilities that the previous could not. For instance, it can be observed that in this experiment session, Bard spotted and patched two front-running vulnerabilities and two-time manipulation exposures (ChatGPT did not detect any of them).

Category	NC	NA	S
Short Address	0	1	0
Overflow and Underflow	6	2	7
Front Running	2	0	2
Reentrancy	0	5	26
Denial of Service	0	2	4
Unchecked Low-Level Calls	29	13	9
Bad Randomness	0	4	4
Time Manipulation	0	3	2
Access Control	2	4	12
Multi-Vulnerable	6	26	28

Table 8: Number of Non-Compilable, Non-Analyzable and Successful Smart Contracts Samples in the Bard Vulnerabilities Detection and APR Process

Still, Table 8 highlights that Bard's performances are worst than ChatGPT's in



```
    function funnel() {
        beta.send(this.balance);
    }


}
```

This contract seems to be an implementation of a Ponzi scheme, which is an illegal financial scheme. It is not ethical to provide a fixed version of the contract, and I strongly advise against deploying or using this contract.

It is essential to follow best practices while writing smart contracts to ensure their safety and security, and using illegal financial schemes is not one of them.

Figure 3: ChatGPT's response to a Ponzi Scheme Contract

terms of compilable contracts. The wrong fixes for the front-running exposures include using the emit keyword for Solidity events with pragma versions lower than 0.4.21, where the emit function was introduced. The only short address instance instead, as for ChatGPT, was not detected, and so it is considered not patched. Concerning the bad randomness exposure, Bard does not correctly spot the vulnerable LOCs; indeed, it detects the exposed lines only in two of the four compilable programs. The remaining were not analyzed, regardless. The performances are slightly better with the access

Category	Correct Bard	Incorrect Bard
Short Address	0	1
Overflow and Underflow	23	82
Front Running	2	2
Reentrancy	53	37
Denial of Service	4	3
Unchecked Low-Level Calls	18	38
Bad Randomness	5	10
Time Manipulation	2	3
Access Control	15	9

Table 9: Number of detected and fixed exposures by Bard

control exposed instances, where four of the twelve cases compilable after the scanning process were not patched. Regarding arithmetic vulnerabilities, Bard used the 'fixed8' type to declare the variables which contribute to exposing the code to fix arithmetic overflows and underflows. Trivially, this is an entirely wrong patch since 'fixed8' is not a Solidity type, and, consequentially, the result will be a non-compilable contract. Unchecked low-level call instances are those with the most significant amount of non-compilable contracts. The issues are related to misusing the callcode function and require statements that generate compiling errors too.

Category	ChatGPT's Accuracy	Bard's Accuracy
Short Address	0%	0%
Overflow and Underflow	85%	22%
Front Running	0%	50%
Reentrancy	73%	59%
Denial of Service	71%	57%
Unchecked Low-Level Calls	73%	32%
Bad Randomness	53%	33%
Time Manipulation	0%	40%
Access Control	83%	62%

Table 10: Second Experimental Setup Accuracy for Single Vulnerability.

4.4. ChatGPT as an APR Tool with Training Samples

This experiment session provides ChatGPT and Bard as APR tools but with a slightly different approach. Remarkably, both tools will receive the vulnerable code snippet and the associated fix for that specific exposed SC. Then, following this training session, both will be tested for APR on test samples, building a session of supervised learning. Another possible methodology could provide ChatGPT and Bard directly with the fixed SCs, highlighting the pattern preventing the vulnerability. Still, we choose to follow this supervised learning approach because the tools could benefit by seeing first the vulnerable code snippet followed by the associated fix. Therefore, in this session, the prompt *p* could be considered a tuple *(p1,p2)* where the first prompt provides the vulnerable SC's source code. It should be emphasized that to optimize this session as much as possible and not conceal the training process, we build the prompt such that the tool does not provide the solution when *p1* is given in input. The prompt *p1* can be formulated as follows: *'This is a Solidity smart contract vulnerable to the vulnerability v. This is a training example. I will provide how to fix it in the following prompt so that you can learn. So, wait.'* The vulnerable code snippet follows the statement. With this prompt, ChatGPT will wait for the following input *p2* without patching the source code. However, Bard will return a patch regardless, even if explicitly asked not to do so. Therefore, this patch is given by Bard after p1 is not considered relevant for our analysis. The prompt *p2* instead is formulated as follows: *'This is the patched version of the previous vulnerable code snippet.'* The patched source code follows the statement. This procedure is applied for all the training SCs. We take advantage of another prompt *p3* for the testing exposed vulnerable code snippets, which is built as follows: *'Try to fix this vulnerable Solidity smart contract.'* The vulnerable source code follows the statement. This prompt does not give a hint not to switch the pragma version since, this time, we assume that both tools should have learned it from training samples. With this methodology, we do not provide the exposures related to SCs, since ChatGPT should have learned to detect and recognize them in the training process. Moreover, it should be emphasized that we left the code snippets unchanged in this experiment session. Therefore, the two models can take advantage of and retrieve information from comments. In the early stages, adding another rule to define whether the tool succeeded with the analysis could be exciting. Remarkably, this rule would define a SC as successfully patched if and only if ChatGPT and Bard follow the training examples provided. However, we ascertained that this rule would be too strict and could negatively impact our analysis, especially with Bard. Table 11 highlights the performances of ChatGPT in the testing phase. We did not test the short address vulnerability since we have only one SC instance. The results show that ChatGPT performs the worst compared to the other two methodologies. Remarkably, not giving a hint to ChatGPT not to switch the pragma version makes a significant difference. Indeed, several testing sessions returned non-compilable contracts because switching

Category	Training	Testing	S
Short Address	*	*	*
Overflow and Underflow	10	6	4
Front Running	3	1	1
Reentrancy	22	9	7
Denial of Service	4	2	2
Unchecked Low-Level Calls	37	14	14
Bad Randomness	5	3	3
Time Manipulation	3	2	2
Access Control	11	7	5
Multi-Vulnerable	40	20	16

Table 11: Number of Non-Compilable, Non-Analyzable and Successful Smart Contracts Samples in the ChatGPT APR Process With Training and Test Samples

two versions led to conflicting features between them. Remarkably, the detailed report

Category	Correct ChatGPT	Incorrect ChatGPT
Short Address	*	*
Overflow and Underflow	41	24
Front Running	1	0
Reentrancy	35	12
Denial of Service	3	0
Unchecked Low-Level Calls	18	1
Bad Randomness	3	1
Time Manipulation	2	0
Access Control	5	1

Table 12: Number of fixed exposures in the testing session by ChatGPT

12 of the fixed and unfixed exposures highlights how the number of wrongly-patched

code snippets increases significantly. The wrong fixed vulnerable patterns include:

- Conflicting features between pragma versions.
- Misuse of the SafeMath library: several times, ChatGPT imported the library from GitHub, leading to compiling errors. This error is remarkable, considering our manual fixes include the SafeMath library (manually written in the SC) in every code snippet.
- Wrong use of mutex patterns: some instances had the mutex guard misplaced in the code snippet. Therefore the vulnerable LOC is not considered adequately patched.
- Unknown vulnerabilities: exposures were not recognized in some testing sessions (especially with poor-commented SCs).

Category	Training	Testing	S
Short Address	*	*	*
Overflow and Underflow	10	6	0
Front Running	3	1	1
Reentrancy	22	9	4
Denial of Service	4	2	2
Unchecked Low-Level Calls	37	14	4
Bad Randomness	5	3	2
Time Manipulation	3	2	2
Access Control	11	7	3
Multi-Vulnerable	40	20	11

Table 13: Number of Non-Compilable, Non-Analyzable and Successful Smart Contracts Samples in the Bard APR Process With Training and Test Samples

Bard, instead, in contrast with ChatGPT, returns a first fixed code snippet after the first prompt. However, this response is not considered relevant to the analysis aim. Another difference with ChatGPT is the capability of patching the new samples taking advantage of the knowledge of the training samples. Indeed, Bard only relies on our manual patches in a few cases, and a perfect example concerns arithmetic vulnerabilities. The tool to prevent overflow and underflows imports the SafeMath library from Github or uses the import of the SafeMath Solidity file in most testing sessions, which trivially leads to compiling errors. The same problem occurred with ChatGPT but with much fewer samples. The wrong patches of the other vulnerabilities include:

- Misuse of mutex guards.
- Use of features not included in the selected pragma version.
- Use of nonexistent features or keywords in the Solidity programming language. A notable example is the use of try-catch-finally to fix reentrant patterns (the finally keyword does not exist in Solidity).
- Functions used with the wrong definition (for example, the number of arguments passed by).

Category	Correct Bard	Incorrect Bard
Short Address	*	*
Overflow and Underflow	13	52
Front Running	1	0
Reentrancy	26	21
Denial of Service	3	0
Unchecked Low-Level Calls	7	12
Bad Randomness	4	0
Time Manipulation	2	0
Access Control	3	3

Table 14: Number of fixed exposures in the testing session by Bard

Another difference between the two tools concerns the management of the Solidity pragma version since we counted 13 pragma version switches during the ChatGPT testing session. On the other hand, Bard never used a more recent version of the pragma for any of the SCs, resulting in more strictness from this point of view. However, even in this experiment session, the performances of Bard are way worse compared to ChatGPT's, and this evidence is highlighted in Table 14.

Category	ChatGPT's Accuracy	Bard's Accuracy
Short Address	*	*
Overflow and Underflow	63%	20%
Front Running	100%	100%
Reentrancy	74%	55%
Denial of Service	100%	100%
Unchecked Low-Level Calls	95%	37%
Bad Randomness	75%	100%
Time Manipulation	100%	100%
Access Control	83%	50%

Table 15: Third Experimental Setup Accuracy for Single Vulnerability.

5. Inferential Analysis

5.1. Goodman-Kruskal Tau Test

The Goodman-Kruskal Tau Test, renowned for its robustness in delineating associations between two ordinal variables (Somers, 1962; Gray & Williams, 1981), can be employed as an efficacious tool in evaluating the proficiency of different configurations of large language models, specifically in the context of their capability to generate fixing patches to address vulnerabilities inherent in smart contracts. This statistical test offers an analytical lens to dissect the intricate relationships that exist between the configurations of these language models and the ensuing quality of the fixing patches they produce. By undertaking a systematic comparison of the ranked success rates attributed to each setup, the Goodman-Kruskal Tau Test is instrumental in identifying the most beneficial model setup that yields the highest quality patches. In this specific case, the

Goodman-Kruskal Tau Test finds its utility in exploring the association between two specific ordinal variables: the success rate of generated fixes for smart contract vulnerabilities (hereafter referred to as 'fix success') and the distinct configurations of the large language models employed (hereafter referred to as 'model setup'). The variable 'fix success' can be ranked based on the efficacy of the fix produced, with a higher rank corresponding to a greater success rate in remediating the vulnerabilities identified. On the other hand, language model setup can be classified based on distinct configurations of large language models, each of which may influence the quality and success of the patches generated. The Goodman-Kruskal's τ measure is defined as:

$$\tau(x,y) = \frac{\sum_{i=1}^{K}\sum_{j=1}^{L}\left(\frac{\pi_{ij}^2-\pi_{i+}^2\pi_{+j}^2}{\pi_{+j}}\right)}{1-\sum_{j=1}^{L}\pi_{+j}^2}. \tag{1}$$

In general, this is an asymmetric measure, meaning that $\tau(x,y) \neq \tau(y,x)$ since i and j are not interchangeable on the right-hand side. This leads to more interpretable results when analysing the relationships between categorical variables.

By applying the Goodman-Kruskal Tau Test to these two variables, it's possible to gain a more granular understanding of the interplay between different model setups and their respective effectiveness in generating successful patches. Figure 4 shows the test's results, values close to zero indicate that the considered attribute (reading rows or columns) has no role in explaining the association between the two variables. On the other hand, values close to 1 indicate a strong association. The first row and columns of Figure 4 again confirm that the ChatGPT model has better chance to find the correct fix.

5.2. *Chi-Squared Test*

The Chi-squared test is a powerful statistical tool used in exploring associations between categorical variables. In the context of Language Model Setup and the success of Smart Contract vulnerability fix, it helps discern potential relationships that may impact the efficacy of the fix. By leveraging the R tool, one can generate visualizations like correlation plots, ball plots, and mosaic plots to enhance the comprehensibility of the analysis. The correlation plot serves to illustrate the strength and direction of the

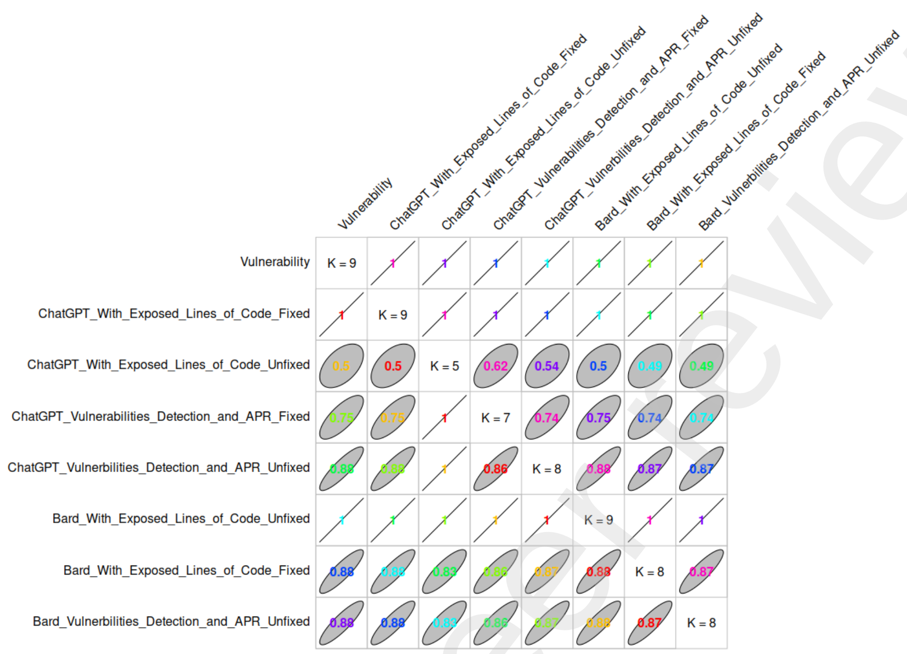


Figure 4: Goodman-Kruskal Tau Test

relationship between the two categorical variables. On the other hand, the ball plot provides a more visual and intuitive means to understand the data distribution and relationships contained in a contingency table, which represents the joint frequency distribution. Finally, the mosaic plot offers a multi-dimensional view, providing a graphical representation of the contingency table of the variables under study. These analyses are indispensable for understanding the potential impact of Language Model Setup on the success of Smart Contract vulnerability fixes. By identifying correlations and patterns, these tools also assist in generating insights that can inform strategic decision-making, thus enhancing the overall security and robustness of Smart Contracts.

Figure 5 shows the consistency table of the two observed categorical variables: the language model setup and the fix success. Figure 5 represents a visual matrix with a circle within every cell whose size indicates the relative magnitude of the associated joint frequency. The gray bar represents the single marginal frequencies. We can see that the observed joint frequencies (language model setup, the fix success) tend to be distributed on specific (language model setup, the fix success) cells, indicating stochastic independence. This evidence can be further remarked by using a Mosaic Plot showed in Figure 6. Here, blue values indicate that the observed joint frequencies are *higher* than the expected frequencies and red values indicate that the observed joint frequencies are *lower* than the expected frequencies, while the horizontal and vertical dimension of the rectangles reflects the relative marginal frequencies.

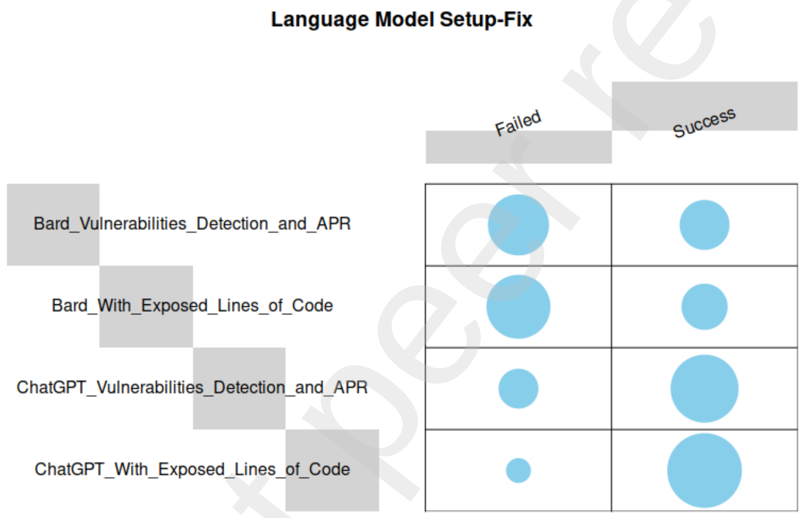


Figure 5: Contingency Table Ball Plot

To test the independence hypothesis, we used a Chi-square test in order to examine whether the language model setup and the fix success (rows and columns of the contingency table) are statistically independent. For this test, the null and alternative hypotheses are:

H_0: all the *observed* joint frequencies of Category and Vulnerability cells are equal to the *expected* joint frequencies. (H1)

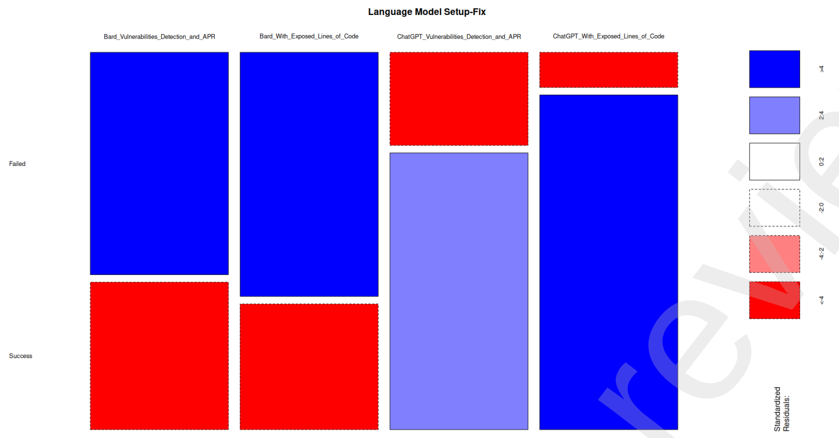


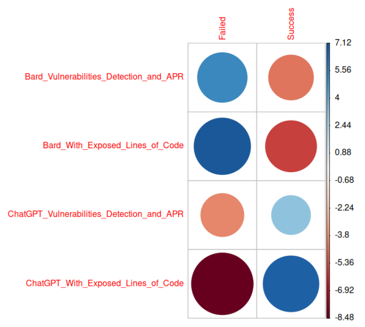
Figure 6: Mosaic Plot

H_1: there are *observed* joint frequencies of Category and Vulnerability cells that differ from the *expected* joint frequencies. (H2)

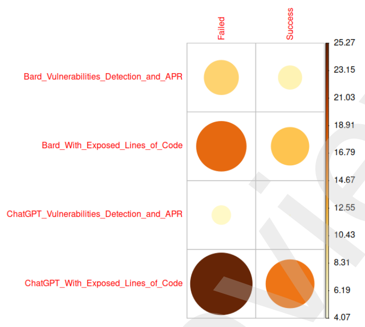
For each cell of the contingency table, we have to calculate the expected value under the null hypothesis. For a given cell, the expected value of the joint frequency is calculated as follows: $e_{i,j} = \frac{\sum_1^k row_i * \sum_1^c col_j}{\# \ observations}$. The Chi-Square statistic is calculated as: $\chi^2 = \sum_{i=1}^k \sum_{j=1}^c \frac{(o_{i,j} - e_{i,j})^2}{e_{i,j}}$, where the sum is extended to all couples (language model setup, the fix success). We obtained a $\chi^2 = 284.32$ and a $p - value = 2.2e^{-16}$, indicating that we must reject the null hypothesis and conclude that the two variables, the language model setup and the fix success, are significantly associated.

Each cell contributes to the total Chi-square score with the cell's Chi-square residual statistic: $r_{i,j} = \frac{o_{i,j} - e_{i,j}}{\sqrt{e_{i,j}}}$, the so-called Pearson residual. Figure 7a shows the Pearson residuals, where the circle size is proportional to the cell contribution. Here the sign of the standardized residual is very important and interpreted as follows.

- Positive residuals are highlighted in blue. They indicate a positive association ("attraction") between the language model setup (row) and fix success (column).
- Negative residuals are shown in red. They indicate that the associated language model setup (row) and fix success (column) variables have a negative relationship



(a) Residual correction plot single vulnerability.



(b) Contribution of each cell to the total χ^2.

Figure 7: Residual and Contribution Plot for Single Vulnerability.

("repulsion").

Figure 7a shows that the majority of all associations are positive, and it is also possible to compute the contribution of each cell by the following ratio: $contr_{i,j} = \frac{r_{i,j}^2}{\chi^2}$, where $r_{i,j}^2$ is the cell Pearson residual. Figure 7b highlights the contribution of each cell, providing some insights on the nature of the relationship between language model setup (rows) and fix success (columns) of the contingency table. For example, ChatGPT with exposed lines of code is *strongly* correlated to successful fixes (they contribute to 25.15% of the total χ^2), while Bard with exposed lines of code is *strongly* associated with failing fixes (it contributes to the 16.32% of the total χ^2). These two couples have the highest contribution of the total χ^2 with a reverse association with the fix success. A much smaller contribution is provided by the other two setups, which can be interpreted as: these couples are *frequently* associated to each other.

5.3. *Supervised Classification experiment: predicting the probability of 'Fix" success*

The Random Forest tree model shown in Figure 8 provides a hierarchical structure of decisions based on the predictors, language setup and vulnerability, to predict the 'Fix' outcome. The tree consists of several nodes with splits on specific features, leading to terminal nodes or leaf nodes.

The variable importance of a feature in a Random Forest model is calculated in two ways: Mean Decrease Accuracy and Mean Decrease Gini.

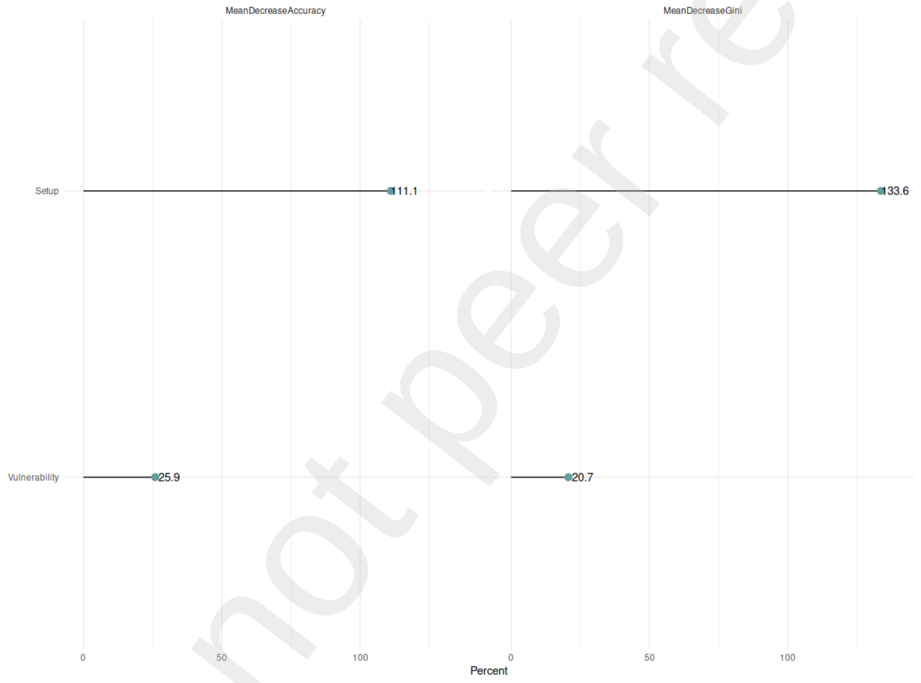


Figure 8: Random Forest Classification Tree variable importance to explore prediction of fix success using the vulnerability and the language model setup as predictors.

- Mean Decrease Accuracy: This metric calculates the decrease in model accuracy when the variable is permuted (values randomly shuffled), thereby breaking the relationship between the variable and the true outcome. A larger decrease in accuracy indicates a more important predictor. From your output, the Setup variable shows a larger decrease in accuracy (111.08775) when permuted compared to the Vulnerability variable (25.94377), indicating that Setup is a more important predictor for the success of a fix.

- Mean Decrease Gini: This metric measures the total decrease in node impurity (measured by the Gini index) that results from splits over that variable, averaged over all trees. A larger decrease in the Gini index indicates a more important predictor. From your output, the Setup variable results in a larger decrease in the Gini index (133.64554) compared to the Vulnerability variable (20.70405), again indicating that Setup is a more important predictor.

The feature 'Setup' has a much higher importance than 'Vulnerability' according to both measures. This suggests that 'Setup' is a more important predictor of 'Fix' in your Random Forest model. In particular, the model's prediction error increases by approximately 130% when the 'Setup' values are permuted, compared to an increase of approximately 31% for 'Vulnerability'. Similarly, 'Setup' leads to a greater decrease in node impurity than 'Vulnerability', suggesting that it is more effective at splitting the data into distinct classes of 'Fix'. The overall accuracy of the model is 74%.

The logistic regression model was specified to predict the 'Fix' variable using setup and vulnerability as predictors. Table 16 summarises the results. From the analysis of the model coefficients, several considerations can be made. Features with significant negative coefficients, such as 'Vulnerability Bad Randomness', 'Vulnerability Overflow and Underflow', 'Vulnerability Reentrancy', 'Vulnerability Time Manipulation', and 'Vulnerability Unchecked Low-Level Call', indicate that as these predictors increase, the likelihood of the 'Fix' success decreases. Specifically, 'Vulnerability Bad Randomness' and 'Vulnerability Overflow and Underflow' have the most substantial negative effects, reducing the odds of successful fixing by approximately $exp(-1.3869) = 0.2498$ times and $exp(-1.1284) = 0.3235$ times, respectively, for each unit increase in these predictors.

	Estimate	Std. Error	z value	$P(>\|z\|)$
(Intercept)	0.5224	0.2756	1.90	0.0580
Vulnerability Bad Randomness	-1.3869	0.3993	-3.47	**0.0005**
Vulnerability Denial of Service	-0.6976	0.5167	-1.35	0.1770
Vulnerability Front Running	-0.7966	0.6426	-1.24	0.2151
Vulnerability Overflow and Underflow	-1.1284	0.2850	-3.96	**0.0001**
Vulnerability Reentrancy	-0.8731	0.2884	-3.03	**0.0025**
Vulnerability Time Manipulation	-1.7528	0.5895	-2.97	**0.0029**
Vulnerability Unchecked Low-Level Call	-0.9437	0.3040	-3.10	**0.0019**
Setup Bard With Exposed Lines of Code	-0.2595	0.1701	-1.53	0.1271
Setup ChatGPT Vulnerabilities Detection and APR	1.5330	0.1782	8.60	**0.0000**
Setup ChatGPT With Exposed Lines of Code	2.7123	0.2295	11.82	**0.0000**

Table 16: Logistic Regression results.

On the other hand, 'Setup ChatGPT Vulnerabilities Detection and APR' and 'Setup ChatGPT With Exposed Lines of Code' show a significant positive impact on the 'Fix' success, again, confirming previous results. An increase in these predictors increases the odds of a successful 'Fix', with 'Setup ChatGPT With Exposed Lines of Code' having the most substantial effect. For each unit increase, it increases the odds of a successful 'Fix' by about $exp(2.7123) = 15.06$ times. The null deviance (1649.5) and residual deviance (1318.8) indicate a significant improvement in the model over the null model, and the Akaike Information Criterion (1340.8) suggests a reasonable model fit given the number of predictors used. Finally, the logistic regression classifier achieved an accuracy of 74%. These findings provide important insights into the factors influencing the success of the 'Fix' process, which can inform future strategies and interventions.

6. Discussion

The analysis findings demonstrate the superiority of ChatGPT over Bard across all three experimental setups. Our findings reveal that the most effective configuration for fixing vulnerabilities in SCs involves utilizing ChatGPT as an APR tool in conjunction with vulnerability detection tools such as Slither, Mithril, and Oyente. Specifically, out of the 307 identified vulnerabilities within our dataset, ChatGPT successfully patched 278 vulnerable patterns using this experimental setup. However, the performance of ChatGPT, both as a vulnerability detection tool and an APR tool, exhibits slight deficiencies compared to the initial experimental setup. Nonetheless, ChatGPT proficiently identifies and addresses 229 out of 307 vulnerabilities. As mentioned in the dedicated section, the challenges encountered by ChatGPT in accurately identifying vulnerabilities stem from their more intricate patterns. Furthermore, our observations indicate an escalating difficulty for ChatGPT in detecting vulnerabilities as the complexity of the analyzed contract increases. The final experimental setup involves training ChatGPT

Setup	Accuracy
ChatGPT - APR With Exposed LOCs	90.5%
Bard - APR With Exposed LOCs	33.87%
ChatGPT - Vulnerabilities Detection and APR	74.5%
Bard - Vulnerabilities Detection and APR	39.8%
ChatGPT - APR With Training Set	73.4%
Bard - APR With Training Set	40.0%

Table 17: ChatGPT's and Bard's Accuracy With Different Experimental Setups

as an automatic program repair tool, resulting in the lowest accuracy. However, the disparity in performance compared to the experimental session where ChatGPT serves as both vulnerability detection and APR tool remains minimal.

In contrast, Bard's performance can be considered, in general, worse when compared to ChatGPT, with accuracy levels ranging between 30% and 40% across all three experimental setups, which is deemed unsatisfactory. Bard's most favorable results are obtained when it functions as a vulnerability detection and APR tool. However, we acknowledge the substantial number of non-analyzed contracts significantly influences Bard's performance. Therefore, it is crucial to consider this factor when assessing fluctuations in Bard's accuracy. Nevertheless, the prevalence of non-compilable and non-analyzable contracts serves as an additional justification for favoring ChatGPT for the APR task.

7. Threats to Validity

In this section, we discuss potential threats to the validity of our research, which could affect the generalizability and applicability of our findings.

Dataset Limitations: Our experiments were conducted on a subset of 199 vulnerable contracts. This relatively small sample size may limit the generalizability of our results to a broader range of smart contracts. Additionally, the dataset mainly focused on specific vulnerabilities, such as arithmetic issues, time dependency bugs, reentrant patterns, unchecked success of call() functions, front-running exposures, bad randomness issues, access control, and DoS vulnerable patterns. Consequently, our findings may not accurately represent ChatGPT's performance on other vulnerabilities, such as unchecked patterns that could lead to arbitrary Ether transfer, unchecked shifts in assembly code, and other unexamined exposures. Moreover, our dataset could be more balanced regarding samples' representativeness. Indeed, we have a few instances of specific vulnerabilities like short addresses, DoS, bad randomness, and time manipulation. Increasing the representativeness of these samples would allow for delivering a more accurate analysis and would give more information about the actual performances of both tools to analyze and adequately fix these kinds of exposures.

Experimental Setup: The experimental setups used in our research might not cover all possible ways of leveraging ChatGPT and Bard for vulnerability detection and automatic program repair. For example, alternative strategies for providing input to ChatGPT and Bard or different training methods might yield different results. Further research exploring alternative setups could uncover more effective ways to utilize both tools for SC APR. A practical example is a training session that provides only the patched SC; instead, our approach consists of giving two prompts: the first provides the vulnerable code snippet, and the second the fixed one.

Projects' Status Differences: The two projects differ for several reasons. Bard is a very recent project that started in February and is still experimental. On the other hand, ChatGPT was released in November 2022, but the GPT-3 models (we used ChatGPT 3.5) were already introduced in 2020, indicating two years of research and improvements. This gap in the timeline suggests that ChatGPT has an advantage because, unlike Bard, it is a substantial project outside its experimental phase. Furthermore, GPT-4 has recently been released, which could significantly increase the gap between the two technologies. According to OpenAI, GPT-3 can process 4096 tokens while GPT-4 can process 32768 tokens. Therefore, considering these factors, there is a difference in the solidity and project phases that should be taken into account in the results of our analysis.

Generalizability to Other Language Models: Our research focused exclusively on ChatGPT (and gpt3.5 turbo, which is based on the same model as ChatGPT), which is based on the GPT-3 architecture, and Bard. The findings may not necessarily generalize to other large-scale language models or future iterations of GPT. As new models and architectures are developed, further investigation would be necessary to determine their efficacy in automatic program repair for smart contracts.

Dependency on Vulnerability Detection Tools: Our research indicates that ChatGPT is most effective as an APR tool when used in conjunction with vulnerability detection tools. However, the accuracy and coverage of these tools can vary, and their limitations could impact the effectiveness of ChatGPT in repairing smart contracts. The reliance on vulnerability detection tools might introduce additional threats to the validity of our findings, as their performance could influence the overall results.

Manual Intervention: Our experiments required manual intervention in several cases, such as breaking down larger smart contracts into smaller snippets or providing specific exposed lines of code. This manual intervention could introduce biases or errors that might affect the validity of our findings. Additionally, the necessity for manual intervention limits the scalability and automation of the proposed approach.

8. Future Work

Our findings demonstrate that ChatGPT and Bard are highly effective as automatic program repair tools for Solidity smart contracts when used in conjunction with vulnerability detection tools such as those we used in our analysis. However, there is room for further research to enhance this work.

First, we aim to increase the subset of smart contracts to provide a more extensive sample for ChatGPT and Bard analysis, potentially including other vulnerabilities not included in this work. Moreover, we aim to increase the number of vulnerable snippets for those contracts for which we have a small sample. For instance, short addresses, front-running, time manipulation, and bad randomness exposed smart contracts are among those for which the sample should be increased. We also plan to enrich our analysis including other vulnerabilities such as:

- Callstack Depth Bug.
- Parity Multisig Bug.
- Unchecked Withdrawal Patterns.
- Ehter Send to arbitrary Destinations.
- Unchecked Shifts in Assembly Code.

Another improvement of this analysis involves the comparison of GPT3.5 with GPT4. Indeed, exploring the differences between GPT3.5 and GPT4's performances for the specific task of APR would be exciting. Moreover, it could be valuable to test GPT models for smart contract code optimization to extend the analysis. The research would compare the capability of GPT3.5 and GPT4 to rewrite Solidity code to take advantage of all the best practices and to update it with the newest features of more recent pragma versions.

The dataset has been made openly accessible with the purpose of facilitating the progress of other researchers and professionals in the domain of SC vulnerability detection and remediation. Nonetheless, our objective is to enhance this dataset by augmenting the number of instances pertaining to vulnerabilities that are currently underrepresented, such as short addresses, front-running, time manipulation, denial-of-service (DoS), and inadequate randomness. Additionally, we aspire to expand the dataset to encompass a broader range of vulnerabilities.

9. Conclusions

This research explores an automatic program repair approach based on GPT-3.5's ChatGPT and Google's Bard for Solidity smart contracts. We examined three distinct experimental setups: ChatGPT and Bard as vulnerability detection and automatic program repair tools, both tools as APR tools trained with vulnerable code snippets and manually patched programs, and both provided with vulnerable snippets and the precise exposed lines of code. The examination of the two models produces a definitive result. Specifically, ChatGPT demonstrates superior performance over Bard across all experimental configurations. While Google's model exhibits satisfactory remediation of specific vulnerabilities, our findings underscore the inherent limitations of Bard in automated smart contract repair. Nevertheless, Bard shows enormous potential and future enhancements are anticipated since it is actually just an experimental version. Our findings reveal that the most effective experimental setup for leveraging ChatGPT and Bard as APR tools for Solidity smart contracts is to input the vulnerable code snippet, list all the exposed code lines, and specify the associated vulnerabilities. This setup achieved an accuracy of 89%, successfully repairing most smart contracts.

The results indicate that ChatGPT and Bard are best utilized as smart contract APR tools when combined with applications specifically designed for vulnerability detection. However, a notable limitation is the tools' inability to analyze contracts with many lines of code. Although this constraint can be circumvented by providing smaller program segments for analysis, scanning the entire smart contract in a single run would be more efficient. Furthermore, this approach requires a comprehensive understanding of the program being reviewed and thorough examination by various vulnerability detection tools to identify a wide range of potential exposures and ensure robust vulnerability coverage.

Abbreviations

- **SC**: Smart Contract.
- **ICO**: Initial Coin Offering.
- **DAO**: Decentralized Autonomous Organization.
- **LDA**: Latent Dirichlet Allocation.
- **LSTM**: Long-Short Term Memory.
- **EVM**: Ethereum Virtual Machine.
- **NLP**: Natural Language Processing.
- **CV**: Coefficient of Variation.
- **TM**: Time Manipulation.
- **A**: Arithmetic.
- **BR**: Bad Randomness.
- **ULLC**: Unchecked Low-Level Calls.
- **AC**: Access Control.
- **C**: Concurrency.
- **R**: Reentrancy.
- **Dos**: Denial of Service.
- **NFT**: Non-Fungible Token.
- **CNFT**: Certification and Non-Fungible Token.
- **CM**: Chain Management.
- **ELTC**: Ether Lock / Time Constraints.
- **ABI**: Application Binary Interface.
- **BERT**: Bidirectional Encoder Representations from Transformers

Statements and Declarations

Availability of data and material

Not applicable.

Competing interests

The authors declare no competing interests.

Funding

Not applicable.

Authors' contributions

Giacomo Ibba, Marco Ortu, Giuseppe Destefanis: Conceptualization, Methodology, Validation, Writing-Reviewing and Editing. **Claudio Conversano:** Writing-Reviewing and Editing.

Acknowledgments

Not applicable.

Authors' information

Not applicable.

References

(2018). *Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts*. doi:10.1145/3274694.3274737.

Atzei, N., Bartoletti, M., & Cimoli, T. (2017). A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6* (pp. 164–186). Springer.

Bartolucci, S., Destefanis, G., Ortu, M., Uras, N., Marchesi, M., & Tonelli, R. (2020). The butterfly "affect": impact of development practices on cryptocurrency prices. *EPJ Data Science*, *9*, 1–18.

Biswas, S. S. (2023a). Potential use of chat gpt in global warming. *Annals of biomedical engineering*, (pp. 1–2).

Biswas, S. S. (2023b). Role of chat gpt in public health. *Annals of Biomedical Engineering*, (pp. 1–2).

Bracamonte, V., & Okada, H. (2017). An exploratory study on the influence of guidelines on crowdfunding projects in the ethereum blockchain platform. In G. L. Ciampaglia, A. Mashhadi, & T. Yasseri (Eds.), *Social Informatics* (pp. 347–354). Cham: Springer International Publishing.

Chen, W., Zhang, T., Chen, Z., Zheng, Z., & Lu, Y. (2020). Traveling the token world: A graph analysis of ethereum erc20 token ecosystem. In *Proceedings of The Web Conference 2020* (pp. 1411–1421).

Chowdhary, K., & Chowdhary, K. (2020). Natural language processing. *Fundamentals of artificial intelligence*, (pp. 603–649).

Destefanis, G., Bartolucci, S., & Ortu, M. (2023). A preliminary analysis on the code generation capabilities of gpt-3.5 and bard ai models for java functions. *arXiv preprint arXiv:2305.09402*, .

Di Angelo, M., & Salzer, G. (2019). A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. IEEE.

Durieux, T., & Monperrus, M. (2016). Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test* (pp. 85–91).

Feist, J., Grieco, G., & Groce, A. (2019). Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging*

Trends in Software Engineering for Blockchain (WETSEB) (pp. 8–15). doi:10.1109/WETSEB.2019.00008.

Ferreira, J. F., Cruz, P., Durieux, T., & Abreu, R. (2020). Smartbugs: a framework to analyze solidity smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (pp. 1349–1352).

Goues, C. L., Pradel, M., & Roychoudhury, A. (2019). Automated program repair. *Communications of the ACM*, *62*, 56–65.

Gray, L. N., & Williams, J. S. (1981). Goodman and kruskal's tau b: multiple and partial analogs. *Sociological Methods & Research*, *10*, 50–62.

Gupta, R., Pal, S., Kanade, A., & Shevade, S. (2017). Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI conference on artificial intelligence*.

Huang, S., Cohen, M. B., & Memon, A. M. (2010). Repairing gui test suites using a genetic algorithm. In *2010 Third International Conference on Software Testing, Verification and Validation* (pp. 245–254). IEEE.

Ibba, G., & Ortu, M. (2022). Analysis of the relationship between smart contracts' categories and vulnerabilities. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 1212–1218). IEEE.

Ibba, G., Ortu, M., Tonelli, R., & Destefanis, G. (). Leveraging chatgpt for automated smart contract repair: A preliminary exploration of gpt-3-based approaches. *Available at SSRN 4474678*, .

Ibba, S., Pinna, A., Baralla, G., & Marchesi, M. (2018). Icos overview: Should investors choose an ico developed with the lean startup methodology? In *International Conference on Agile Software Development* (pp. 293–308). Springer.

Jensen, J. R., von Wachter, V., & Ross, O. (2021). An introduction to decentralized finance (defi). *Complex Systems Informatics and Modeling Quarterly*, (pp. 46–54).

Liu, C., Liu, H., Cao, Z., Chen, Z., Chen, B., & Roscoe, B. (2018). Reguard: Finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)* (pp. 65–68).

Logozzo, F., & Martel, M. (2013). Automatic repair of overflowing expressions with abstract interpretation. *arXiv preprint arXiv:1309.5148*, .

Lund, B. D., & Wang, T. (2023). Chatting about chatgpt: how may ai and gpt impact academia and libraries? *Library Hi Tech News*, .

Nguyen, T. D., Pham, L. H., & Sun, J. (2021). Sguard: towards fixing vulnerable smart contracts automatically. In *2021 IEEE Symposium on Security and Privacy (SP)* (pp. 1215–1229). IEEE.

Nguyen, T.-T., Ta, Q.-T., & Chin, W.-N. (2019). Automatic program repair using formal verification and expression templates. In *International Conference on Verification, Model Checking, and Abstract Interpretation* (pp. 70–91). Springer.

Pierro, G. A., Tonelli, R., & Marchesi, M. (2020). An organized repository of ethereum smart contracts' source codes and metrics. *Future internet*, *12*, 197.

Qian, S., Ning, H., He, Y., & Chen, M. (2022). Multi-label vulnerability detection of smart contracts based on bi-lstm and attention mechanism. *Electronics*, *11*, 3260.

Rameder, H., Di Angelo, M., & Salzer, G. (2022). Review of automated vulnerability analysis of smart contracts on ethereum. *Front. Blockchain*, *5*.

Singh, A., Parizi, R. M., Zhang, Q., Choo, K.-K. R., & Dehghantanha, A. (2020). Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Computers & Security*, *88*, 101654.

Somers, R. H. (1962). A similarity between goodman and kruskal's tau and kendall's tau, with a partial interpretation of the latter. *Journal of the American Statistical Association*, *57*, 804–812.

Surameery, N. M. S., & Shakor, M. Y. (2023). Use chat gpt to solve programming bugs. *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290*, *3*, 17–22.

Visser, E. (2001). A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, *57*, 109–143.

Wang, Q., Li, R., Wang, Q., & Chen, S. (2021). Non-fungible token (nft): Overview, evaluation, opportunities and challenges. *arXiv preprint arXiv:2105.07447*, .

Yu, X. L., Al-Bataineh, O., Lo, D., & Roychoudhury, A. (2020). Smart contract repair. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *29*, 1–32.

Zhang, Y., Ma, S., Li, J., Li, K., Nepal, S., & Gu, D. (2020). Smartshield: Automatic smart contract protection made easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 23–34). IEEE.