

Titre: Toward Efficient Smart Contract Auditing: Integrating Large
Title: Language Models for Automated Reporting and Adaptive Test Case
Generation

Auteur: Mohamed Salah Bouafif
Author:

Date: 2024

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Bouafif, M. S. (2024). Toward Efficient Smart Contract Auditing: Integrating Large
Citation: Language Models for Automated Reporting and Adaptive Test Case Generation
[Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie.
<https://publications.polymtl.ca/61716/>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/61716/>
PolyPublie URL:

**Directeurs de
recherche:** Mohammad Hamdaqa
Advisors:

Programme: Génie informatique
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

**Toward Efficient Smart Contract Auditing: Integrating Large Language Models
for Automated Reporting and Adaptive Test Case Generation**

MOHAMED SALAH BOUAFIF

Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Décembre 2024

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

**Toward Efficient Smart Contract Auditing: Integrating Large Language Models
for Automated Reporting and Adaptive Test Case Generation**

présenté par **Mohamed Salah BOUAFIF**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Maxime LAMOTHE, président

Mohammad HAMDAQA, membre et directeur de recherche

Heng LI, membre

DEDICATION

For my family and friends . . .

ACKNOWLEDGEMENTS

As I navigate the complexities of scientific research, I would like to express my deepest gratitude to the individuals who have supported me throughout my Master's journey.

First and foremost, I would like to thank my supervisor, Dr. Mohammad Hamdaqa, for his guidance, expertise, and unwavering support. Your mentorship has been instrumental in shaping my research and academic growth.

I extend my sincere appreciation to Dr. Ed Zulkoski, Head of Research at Quantstamp. Inc, for providing invaluable feedback and sharing his profound expertise in blockchain technology. The opportunity to work within Quantstamp has not only broadened my knowledge but also deepened my understanding of the field. Your contributions have significantly enhanced the quality of my research. I would also like to thank my colleagues, peers, and friends for their encouragement, discussions, and collaboration throughout this journey.

Lastly, I am grateful to my family and loved ones for their unconditional support, patience, and understanding. Your presence has made this achievement possible.

Thank you all for helping me get through this impactful journey.

RÉSUMÉ

L'adoption croissante de la technologie blockchain a entraîné une augmentation de l'utilisation des contrats intelligents pour l'exécution de transactions de grande valeur et la gestion de systèmes décentralisés. Les contrats intelligents faisant de plus en plus partie intégrante de l'écosystème de la blockchain, il est primordial de garantir leur sécurité. Cependant, la complexité des contrats intelligents, associée au développement rapide de la finance décentralisée (DeFi) et d'autres applications basées sur la blockchain, a créé une demande importante de pratiques de codage sécurisées et d'outils d'audit fiables.

Ce projet de recherche aborde les défis de l'audit des contrats intelligents en développant de nouvelles approches pour aider les auditeurs humains et améliorer l'efficacité et la précision du processus d'audit. Nous présentons deux contributions clés : (1) une technique d'incitation contextuelle pour les grands modèles de langage (LLM) afin de détecter les vulnérabilités dans les contrats intelligents, et (2) une technique de priorisation, PRiMG, pour la génération efficace de cas de test en utilisant des tests de mutation.

Notre technique d'incitation contextuelle améliore la conception d'incitations grâce à trois techniques clés : Le cadrage du code, le cadrage de l'évaluation et le cadrage du rapport. Ces méthodes optimisent le processus d'audit en guidant les LLM pour qu'ils se concentrent sur les aspects les plus pertinents du code. Nous évaluons l'efficacité de notre approche en utilisant des contrats intelligents réels.

PRiMG, notre technique de priorisation, s'appuie sur les tests de mutation pour améliorer l'efficacité des cas de test générés par les LLM pour les contrats intelligents Solidity. PRiMG se compose de deux éléments principaux : un module de hiérarchisation des mutations et un module de génération de cas de test. Nous démontrons l'efficacité de PRiMG en utilisant des projets Solidity réels de Code4Arena.

Cette recherche contribue au domaine de la sécurité des blockchains en fournissant des outils et des méthodologies avancés pour l'audit des contrats intelligents. Notre approche comble le fossé entre l'expertise humaine et les outils automatisés, garantissant que les contrats intelligents sont à la fois sécurisés et dignes de confiance dans les applications du monde réel. L'adoption généralisée des technologies blockchain dans les industries dépend de la fiabilité et de la sécurité des contrats intelligents, ce qui rend cette recherche opportune et impactante.

ABSTRACT

The increasing adoption of blockchain technology has led to a surge in the use of smart contracts for executing high-value transactions and governing decentralized systems. As smart contracts become more integral to the blockchain ecosystem, ensuring their security is paramount. However, the complexity of smart contracts, coupled with the rapid development of decentralized finance (DeFi) and other blockchain-based applications, has created a significant demand for secure coding practices and reliable auditing tools.

This research project addresses the challenges of smart contract auditing by developing novel approaches to assist human auditors and enhance the efficiency and accuracy of the auditing process. We present two key contributions: (1) a context-driven prompting technique for Large Language Models (LLMs) to detect vulnerabilities in smart contracts, and (2) a prioritization technique, PRiMG, for efficient test case generation using mutation testing.

Our context-driven prompting technique enhances prompt design through three key techniques: Code Scoping, Assessment Scoping, and Reporting Scoping. These methods optimize the auditing process by guiding LLMs to focus on the most relevant aspects of the code. We evaluate the effectiveness of our approach using real-world smart contracts.

PRiMG, our prioritization technique, leverages mutation testing to enhance the effectiveness of test cases generated by LLMs for Solidity smart contracts. PRiMG consists of two main components: a mutation prioritization module and a test case generation module. We demonstrate the effectiveness of PRiMG using real-world Solidity projects from Code4Arena.

This research contributes to the field of blockchain security by providing advanced tools and methodologies for smart contract auditing. Our approach bridges the gap between human expertise and automated tools, ensuring that smart contracts are both secure and trustworthy in real-world applications. The widespread adoption of blockchain technologies across industries depends on the reliability and security of smart contracts, making this research timely and impactful.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
RÉSUMÉ	v
ABSTRACT	vi
TABLE OF CONTENTS	vii
LIST OF TABLES	ix
LIST OF FIGURES	x
LIST OF SYMBOLS AND ACRONYMS	xi
CHAPTER 1 INTRODUCTION	1
1.1 Software auditing	1
1.1.1 Smart contract auditing	2
1.1.2 Software Testing	4
1.1.3 LLM for test-case generation for smart contracts	5
1.2 Research statement	6
1.3 Thesis contributions	7
1.4 Thesis structure	8
CHAPTER 2 LITERATURE REVIEW	9
2.1 Smart contracts	9
2.2 DeFi Hacks	10
2.3 Traditional vulnerability detection tools	11
2.3.1 Code Auditing	11
2.3.2 Static analysis based approaches for smart contracts auditing	12
2.3.3 Dynamic analysis and machine learning-based approaches for smart contracts auditing	12
2.4 LLM-based approaches for smart contracts auditing	13
2.5 Automated test case generation for smart contracts	15
2.6 Chapter Summary	16

CHAPTER 3	A CONTEXT-DRIVEN APPROACH FOR CO-AUDITING SMART CONTRACTS USING GPT-4 CODE INTERPRETER	17
3.1	Approach	19
3.1.1	Motivation example	19
3.1.2	Approach	25
3.2	Evaluation	30
3.2.1	Data Sources	31
3.2.2	Experiment Setup	32
3.2.3	Empirical Study Results	33
3.3	Discussions	40
3.4	Threat to Validity	43
3.5	Chapter Summary	45
CHAPTER 4	PRIORITIZING MUTANTS FOR EFFICIENT TEST SUITE GENERATION USING LLM	46
4.1	Approach	48
4.1.1	Test Case Generation Module	48
4.1.2	Prioritization module	51
4.2	Evaluation	54
4.2.1	Experimental Setup	55
4.2.2	Empirical Study Results	56
4.3	Discussions	62
4.3.1	Interpretation of the importance of the loop size on the effectiveness of test case generation	62
4.3.2	Adequacy of our approach for practical applications	63
4.4	Threat to Validity	64
4.5	Chapter Summary	65
CHAPTER 5	CONCLUSION	66
5.1	Summary of Works	66
5.2	Limitations and Future Works	67
REFERENCES		69

LIST OF TABLES

Table 3.1	Experiment Sources	32
Table 3.2	Experimental results on Dataset A : number of detected vulnerable function for each scoping technique	33
Table 4.1	Data sources	56
Table 4.2	Number of correct test cases generated in different projects using dif- ferent loop sizes	58
Table 4.3	Number of killed mutants using Randomly Selected Tests, Prioritized Test, and Ground Truth	62

LIST OF FIGURES

Figure 3.1	Overview of the study method	25
Figure 3.2	Function call graph of the tokenizeVault contract	27
Figure 3.3	CAQ-based prompt	30
Figure 3.4	CWE-based prompt	31
Figure 3.5	Overview of the empirical study approach	31
Figure 3.6	Qualitative evaluation of the GPT generated audits of two reviewers .	40
Figure 4.1	The proposed approach for generating a test case using LLMs.	49
Figure 4.2	Overview of the dataset labeling process	53
Figure 4.3	Number of test cases by trial and projects	59

LIST OF SYMBOLS AND ACRONYMS

LLM	Large Language Model
SC	Smart Contract
AST	Abstract Syntax Tree
CCL	Code Call List
FCG	Function Call Graph
GPT	Generative Pre-trained Transformer
CAQ	Common Asked Questions
CWE	Common Weaknesses Enumeration
DAO	Decentralized Autonomous Organization

CHAPTER 1 INTRODUCTION

Smart contracts have become the foundation of automation and trust through secure and transparent transactions [1]. They are widely adopted in various industries, from finance to real estate to supply chain management, promising efficiency, transparency, and trust in a trustless environment [2, 3]. As a key component of decentralized applications (dApps), smart contracts play a crucial role in enhancing efficiency and reducing operational costs, which makes them increasingly attractive for a wide range of sectors.

A smart contract is essentially a self-executing piece of code on a blockchain, which automatically enforces agreements and executes terms based on predefined conditions. Once deployed, smart contracts operate in a decentralized manner, ensuring that transactions occur as programmed without the need for intermediaries or third-party verification. This immutability and autonomy bring several advantages, but also create certain risks. Unlike traditional software, which can be updated to fix issues, smart contracts are typically immutable, meaning that any vulnerabilities present in the code at deployment remain permanently present, making security a critical concern [4]. The critical importance of smart contract security is underscored by the substantial financial stakes inherent in the DeFi ecosystem, where these contracts often govern assets worth millions or even billions of dollars [5]. The decentralized and autonomous nature of smart contracts precludes any central authority from intervening to reverse or mitigate malicious transactions once vulnerabilities are exploited. This limitation has contributed to several high-profile incidents, including the *625millionRoninBridgeexploit* and the *190 million Nomad Bridge* breach, wherein inadequate security measures rendered smart contracts vulnerable to catastrophic financial losses. These events underscore the necessity for comprehensive security protocols, including meticulous code reviews, formal verification, and third-party audits by experienced professionals. Such proactive measures are not merely protective mechanisms but indispensable practices to foster trust, ensure system reliability, and uphold the integrity of decentralized financial infrastructures.

1.1 Software auditing

Software auditing is a comprehensive process that involves evaluating an organization's software assets, development practices, and usage patterns to ensure quality, security, and compliance. This process typically encompasses several key aspects, including code quality assessment, security vulnerability detection, license compliance verification, and software inventory

management. One crucial component of software auditing is static code analysis, which involves examining the source code without executing it to identify potential bugs, security flaws, and violations of coding standards [6]. Dynamic analysis, on the other hand, involves running the software and observing its behavior to detect run-time errors and performance issues. Advanced software auditing techniques often incorporate machine learning algorithms to improve the accuracy and efficiency of defect detection [6]. Additionally, software auditing may involve reviewing the software development lifecycle (SDLC) processes to ensure adherence to best practices and industry standards. The effectiveness of software audits can be significantly enhanced through the use of automated tools that can analyze large codebases quickly and consistently, providing more timely and transparent reporting.

1.1.1 Smart contract auditing

Smart contract auditing is a specialized form of software auditing that focuses on blockchain-based applications and their underlying code. This process is critical in ensuring the security, functionality, and reliability of smart contracts, which are self-executing contracts with the terms of the agreement directly written into code [7]. Smart contract audits typically follow a structured approach, beginning with a thorough scoping phase to define the audit's objectives and boundaries. The core of the audit involves a detailed review of the contract's code, logic, and potential vulnerabilities, often using a combination of manual inspection and automated tools. Auditors pay particular attention to common vulnerabilities in smart contracts, such as reentrancy attacks, integer overflow/underflow, and gas limit issues [8]. The use of formal verification techniques, which apply mathematical methods to prove or disprove the correctness of algorithms underlying a system, is increasingly being adopted in smart contract auditing to provide higher levels of assurance [9]. After identifying and verifying vulnerabilities, auditors generate comprehensive reports detailing their findings and recommendations for remediation. The smart contract auditing field is rapidly evolving, with ongoing research focusing on developing more sophisticated analysis tools and methodologies to keep pace with the increasing complexity of blockchain applications. Smart contract auditing is typically performed manually by specialized auditing engineers and requires a deep understanding of the business use case and the historical attack events [10]. Auditing smart contracts is time-consuming and challenging, contingent upon the auditor's expertise and experience [10]. Auditors require a diverse skill set that encompasses proficiency in debugging, a deep understanding of various code analysis techniques, knowledge of blockchain platforms (e.g., EVM), and a grasp of the business domain and rules embedded within the protocol's code [10].

Artificial intelligence for smart contracts auditing:

Recently, the application of machine learning and deep learning techniques to analyze Solidity smart contracts has gained significant traction [11, 12]. Deep learning models, especially Recurrent Neural Networks (RNNs) and their variant, Long Short-Term Memory (LSTM) networks, have been employed to capture the sequential nature of code and its inherent patterns [12]. LSTMs, with their ability to remember long-term dependencies, are particularly suited to recognizing intricate patterns in Solidity code that might be indicative of vulnerabilities. While this approach has shown outstanding results, its generalization to unseen smart contracts written in a new version of Solidity remains a concern. Also, these models are not able to generate human-like explanations of the found vulnerabilities and the mitigation steps. On the other hand, GPT has attracted a lot of attention from the research community regarding its ability to code review and generation [13–17]. Particularly in [18], a comprehensive evaluation of GPT-3.5 and GPT-4 code interpreter for vulnerability detection in smart contracts is illustrated. This work suggests using prompt engineering for the auditing of smart contracts. In particular, it first uses GPT to generate the prompt that describes the task of auditing; then, this prompt is used along with the full source code of the contracts as input to GPT. To measure the performance of GPT, the work considered only the Precision, Recall, and F-measure metrics. However, no qualitative evaluation of the generated output was made. The work did not consider the non-deterministic nature of the GPT model regarding the prompt design. Also, this work did not evaluate the capacity of GPT for the task of explaining the found vulnerabilities and the mitigation steps. The experiments showed that the prompt length, i.e., the contract length, can affect negatively the detection performance when exceeding the token limit.

Similarly, in [19] a quantitative evaluation of the performance of GPT-4 code interpreter for the task of vulnerability detection has been performed. Particularly, the prompt design included a description of a specific attack type besides the whole source code of the contract. Common decentralized finance attacks have been classified into 38 Common Weakness Enumeration (CWE), and each smart contract has been prompted along with a description of the specific attack. Although the results of the experiment have shown great potential for GPT-4 code interpreter for detecting vulnerabilities, this experiment has shown that GPT performance is inversely proportional to contract length. The authors in [16] suggest combining fine-tuning and LLM-based agents to detect and explain vulnerabilities in smart contracts. It employs a multi-step process where the model first identifies potential issues, then reasons about the underlying causes, and finally ranks and critiques these explanations to ensure the most accurate and justified results.

In [20], authors focused on using static analysis techniques along symbolic execution for efficient vulnerability detection in smart contracts. By analyzing the inter-contract program dependency graph and performing symbolic execution on the smart contracts, they focus on the detection of reentrancy vulnerabilities. While the evaluation of this approach have shown its efficiency for reentrancy vulnerability detection, the work has not evaluated the approach on other types of smart contracts specific vulnerabilities. The authors in [21] present an innovative framework that utilizes a multi-agent conversational approach for detecting and analyzing vulnerabilities in smart contracts. This approach incorporates two primary strategies: Broad Analysis and Targeted Analysis. Although the evaluation demonstrates enhanced effectiveness in identifying vulnerabilities, the tool is highly resource-intensive.

Additionally, its performance can degrade when analyzing longer code segments. In [22], authors suggest an approach leveraging fine-tuned GPT-4 to identify and fix vulnerabilities in Solidity code by creating an annotated Control Flow Graph (CFG)

1.1.2 Software Testing

Mutation testing

Mutation testing is a powerful technique for evaluating and improving the quality of software test suites. By introducing artificial faults (mutations) into the codebase, it assesses the ability of existing tests to detect these changes, thereby uncovering weaknesses in test coverage and effectiveness [23]. This process is crucial for enhancing software reliability and reducing the risk of undetected bugs in production. However, mutation testing faces significant challenges, particularly in terms of computational resources and scalability. The process requires running the entire test suite against numerous mutated versions of the code, which can be extremely time-consuming and resource-intensive, especially for large codebases or complex applications [23, 24]. This challenge is exacerbated by the generation of a large number of mutants, many of which may be redundant, equivalent, or simply uninteresting, wasting computational resources.

The issue of having too many tests can further compound these challenges. While a comprehensive test suite is generally desirable, it can lead to increased execution times and costs during mutation testing. This is particularly problematic when there are redundant test cases that repeatedly check the same functionality, as it not only increases the time and resources needed for mutation testing but can also make it difficult to interpret results effectively.

Smart contract testing

Smart contract testing is of paramount importance in the blockchain ecosystem due to the immutable and high-stakes nature of these digital agreements. The significance of thorough testing is underscored by the substantial financial losses that can result from vulnerabilities in deployed smart contracts [25]. As these contracts often handle significant monetary transactions and critical business logic, even minor bugs or oversights can lead to catastrophic consequences. Static analysis tools, while efficient, may generate false positives and miss new bug patterns, making dynamic testing techniques a crucial complementary method for comprehensive security assurance [25]. The challenge lies not only in designing effective test oracles and measuring test adequacy but also in automatically generating test data that achieves high coverage [25]. Recent research has explored various approaches, including mutation testing, which involves creating small syntactic changes to evaluate the completeness of test designs [26]. The complexity of smart contracts further necessitates the development of sophisticated testing methodologies, as evidenced by the creation of extensive benchmarks and fine-grained taxonomies of vulnerability types to enhance the evaluation of static application security testing (SAST) tools [26]. As the field evolves, combining multiple testing approaches and tools shows promise in reducing false negatives, though many vulnerabilities remain challenging to detect.

1.1.3 LLM for test-case generation for smart contracts

Recent research has explored the potential of LLMs in generating test cases for smart contracts. One notable approach, PropertyGPT, leverages LLMs to generate comprehensive and high-quality properties for formal verification of smart contracts [27]. This method utilizes a retrieval-augmented property generation technique, embedding existing properties into a vector database and using them for in-context learning to generate new properties for unknown code. Another innovative approach, ConMover, employs a multi-agent framework to tackle the generation of Move smart contracts through a collaborative code generation process [28]. This system mimics human programming approaches by using specialized LLM agents for retrieval, planning, coding, and debugging, operating in a structured pipeline with adaptive traversal. While these approaches show promise, they also face challenges. For instance, iAudit combines fine-tuning and LLM-based agents for intuitive smart contract auditing with justifications [16]. This two-stage approach first tunes a detector model to make decisions and then tunes a Reasoner model to generate causes of vulnerabilities. However, fine-tuning alone faces difficulties in accurately identifying the optimal cause of a vulnerability, necessitating the introduction of additional LLM-based agents for ranking and critiquing. Mutation

testing for generating new test cases presents its own set of challenges, particularly in terms of scalability. Traditional approaches to mutation testing often produce large numbers of redundant mutants, exacerbating computational overheads and human effort in examining live mutants [?]. The generation of high-utility mutations remains a key challenge, with the need to create mutations that are not only syntactically correct but also likely to reveal faults and align with natural coding patterns. Furthermore, determining whether a mutation is equivalent is undecidable, making it challenging to avoid the generation of equivalent mutants and leading to poor scalability [?]. These issues are compounded in the context of smart contracts, where the stakes are high due to the immutable nature of the code deployed and the potential for significant financial losses due to vulnerabilities [29].

1.2 Research statement

Smart contracts represent a critical component of blockchain systems, but their security and reliability are paramount for the ecosystem’s integrity. Building secure, production-level smart contracts presents several challenges that go beyond proof-of-concept solutions. The major challenges these systems encounter are now linked to their deployment and ongoing security rather than just their initial development. Upon reviewing research on smart contract security and the current state of LLM applications in this domain, the following observations can be made:

- Smart contract auditing has become a critical focus for both academia and industry, driven by the increasing financial losses associated with vulnerabilities in deployed contracts. The community is actively developing more sophisticated tools and methodologies to enhance security throughout the entire lifecycle of smart contracts, from initial development to post-deployment monitoring. This is evident in the rise of automated auditing tools that utilize advanced techniques such as static analysis, symbolic execution, and machine learning to detect vulnerabilities efficiently. As the threat landscape continues to evolve, there is a concerted effort to create robust solutions for ongoing monitoring and adaptation of smart contracts in production environments, ensuring their security against emerging threats.
- LLM-based auditing strategies can induce divergent behaviors between human auditors and artificial intelligence-powered tools, potentially causing problems in production. Ensuring the reliability of these strategies thus becomes a critical challenge. Traditional auditing methods, which rely on manual code review and pattern-based analysis, are often not practical because of the increasing complexity of smart contract ecosystems

and the rapid evolution of attack vectors. In contrast, advanced LLM-based approaches can be leveraged to build more comprehensive and adaptable auditing tools that assess the reliability of smart contracts across various scenarios and potential vulnerabilities.

- Mutation testing for smart contracts is a powerful technique for enhancing security, but it demands significant effort from developers. Creating comprehensive test cases to improve mutation scores requires iterative refinement and a deep understanding of both contract logic and blockchain-specific vulnerabilities. Developers must craft tests that not only cover standard programming errors but also address the unique challenges of blockchain environments. While time-consuming, this investment in robust testing ultimately leads to more secure and reliable smart contracts, crucial for high-stakes blockchain applications.

This thesis makes the following contribution to two important aspects related to smart contracts security: (i) A Context-Driven Approach for Co-Auditing Smart Contracts Using GPT-4 Code Interpreter, (ii) A prioritization technique for Efficient Test Suite Generation using LLM. By leveraging the code understanding capabilities of LLMs, this research aims to enhance the detection of vulnerabilities and strengthen the overall security framework for smart contracts. The proposed methodologies focus on automating and refining auditing processes while optimizing testing strategies, thereby improving the reliability and robustness of blockchain-based applications.

1.3 Thesis contributions

In our work, we first develop a co-auditing approach to assist real-world auditors by leveraging the advanced code understanding capabilities of large language models (LLMs). This approach is designed to complement the expertise of human auditors, enabling a more efficient and thorough identification of vulnerabilities in smart contracts. Building on this foundation, we then focus on optimizing the testing process by introducing a tool for efficient test case generation. Our approach consists of two key modules: a test case generation module and a mutant prioritization module. The test case generation module leverages the LLM’s capabilities to generate and refine test cases, ensuring their correctness and effectiveness. The mutant prioritization module efficiently selects mutants as test goals, using a usefulness factor to quantify their potential in enhancing the mutation score. By generating test cases that kill multiple mutants at once, our approach guarantees that the size of the test suite remains optimal while maximizing coverage and improving overall efficiency.

- We propose a novel context-driven prompting technique to enhance the effectiveness

of Large Language Models (LLMs) in smart contract co-auditing, addressing the challenges of context description and code length. By employing Code Scoping to segment code, Assessment Scoping to refine context for targeted vulnerability detection, and Reporting Scoping to standardize outputs, our method achieves a 96% detection rate of vulnerable functions, compared to 53% using native prompts. Empirical evaluations on publicly available vulnerable contracts and expert reviews by professional auditors validate the approach’s reliability and proficiency in unlabeled datasets. This technique not only boosts detection accuracy but also reduces reliance on manual auditing, streamlining the process for greater efficiency and security. This study was submitted to the Information and Software Technologies Journal (2024)

- We propose a novel approach for incremental test suite generation using LLMs, consisting of two key modules: a test case generation module and a mutant prioritization module. The test case generation module leverages the LLM’s understanding capabilities to generate and refine test cases, ensuring their correctness and effectiveness. The mutant prioritization module focuses on efficiently selecting mutants as test goals, using a usefulness factor metric to quantify the potential of a mutant to enhance the mutation score. This approach aims to maximize the number of mutants killed while minimizing the size of the test suite, improving both efficiency and coverage.

1.4 Thesis structure

The remainder of this thesis is organized as follows.

- Chapter 2 provides both background information and literature review on previous related works that are relevant to the main contributions of this thesis.
- Chapter 3 presents our context-driven approach for co-auditing smart contracts, its design, implementation and evaluation.
- Chapter 4 presents our prioritization technique for efficient test case generation using LLMs, its design, implementation, and evaluation on real-world projects.
- Chapter 5 summarizes the thesis and highlights the thesis limitations as well as future work.

CHAPTER 2 LITERATURE REVIEW

In this chapter, we provide all the background information required to introduce our work. Subsequently, we review the literature with regard to the main themes of this research work.

2.1 Smart contracts

Smart contracts and Ethereum have revolutionized the blockchain landscape, introducing a new paradigm for decentralized applications and programmable transactions. Ethereum, launched in 2015, built upon Bitcoin's foundation by incorporating a Turing-complete language, enabling the creation of complex, self-executing contracts. These smart contracts are pieces of code permanently stored on the blockchain, capable of responding to user requests and manipulating digital units of value [30]. The Ethereum platform maintains a global state through a peer-to-peer network of mutually distrusting nodes [30]. This state is stored in a blockchain secured by a proof-of-work consensus mechanism, similar to Bitcoin's [31]. The core innovation of Ethereum lies in its ability to execute complex business logic without relying on trusted third parties, making it appealing for applications in crowdfunding, financial services, identity management, and gambling. Transactions in Ethereum are fundamentally different from those in Bitcoin. While Bitcoin uses an unspent transaction output (UTXO) model, Ethereum employs an account-based model [30]. The state in Ethereum is represented by accounts, each with its own balance, code, and storage [30]. This model allows for more complex interactions and state transitions. The execution of smart contracts in Ethereum is facilitated by the Ethereum Virtual Machine (EVM), a stack-based virtual machine that processes transactions and smart contract code. To address the halting problem and prevent denial-of-service attacks, Ethereum incorporates a pricing mechanism called "gas". Each operation in the EVM consumes a specific amount of gas, making resource-intensive computations economically infeasible. When a transaction is submitted to the Ethereum network, it goes through several states before being processed and confirmed. Initially, the transaction is broadcasted to the network and enters a pending state in the transaction pool. Miners then compete to include the transaction in a block through the Proof-of-Work consensus mechanism. Once included in a block and appended to the blockchain, the transaction transitions to a processed state. The security of smart contracts is a critical concern in the Ethereum ecosystem. As of 2022, out of 19,366 Ethereum contracts examined, 8,833 were found to be insecure, highlighting the prevalence of vulnerabilities. These security issues can lead to significant financial losses, as demonstrated by incidents like the DAO (decentralized

autonomous organization) attack in 2016, which resulted in the theft of approximately 60 million dollars worth of Ether [32]. Smart contract vulnerabilities arise from various sources, including programming errors, logic mistakes, and oversights in security practices. The immutable nature of deployed smart contracts exacerbates these issues, as vulnerabilities cannot be easily patched once the contract is on the blockchain. This immutability, while designed to enhance trust, creates significant challenges in addressing security flaws. The Ethereum Yellow Paper, authored by Gavin Wood, provides a formal definition of the Ethereum protocol. It describes the block structure, state transitions, and the EVM in detail, serving as the authoritative reference for Ethereum’s technical specifications. The Yellow Paper defines the precise mechanics of Ethereum’s operation, including the gas system, transaction execution, and consensus rules. To address the security challenges in smart contracts, researchers and developers have proposed various solutions. These include automated auditing tools, expert reviews, scalable smart contract architectures, and the use of smart contract templates. Additionally, efforts are being made to develop decompilers, semantic frameworks, and anomaly detection systems to enhance smart contract security. As the Ethereum ecosystem continues to evolve, future research directions in smart contract security are likely to focus on improving vulnerability detection techniques, developing more robust programming languages for smart contracts, and enhancing the scalability and performance of decentralized applications. The ongoing transition to Ethereum 2.0, with its shift to a proof-of-stake consensus mechanism, will also introduce new security considerations and research opportunities in the field of smart contracts and blockchain technology [32].

2.2 DeFi Hacks

DeFi (Decentralized Finance) hacks have become a significant concern in the blockchain ecosystem, resulting in substantial financial losses and undermining user trust. According to recent research, these hacks can be categorized into four main types based on the exploited components in DeFi: smart contract exploits, cross-chain bridge exploits, private key leakage, and DeFi front-end attacks [33]. Smart contract exploits are particularly prevalent, having caused at least 1.57 billion USD in losses up to May 1, 2022. These attacks often leverage vulnerabilities such as re-entrancy issues and price oracle manipulation. The DAO attack, which exploited a re-entrancy vulnerability, stands as a notorious example of such exploits. Cross-chain bridge exploits target the mechanisms that facilitate asset exchange between different blockchains, with the Wormhole bridge hack resulting in a loss of 320 million USD. Private key leakage, as seen in the Ronin network attack that led to 625 million USD in losses, highlights the critical importance of secure key management [33].

The rise of private mempool services has added a new layer of complexity to detecting and preventing DeFi attacks. Research shows a significant upward trend in the use of these services by attackers, with the proportion of adversarial transactions sent through private channels rising from the first half of 2022 to a notably higher percentage in the latter half of 2023. This trend underscores the need for new approaches to attack detection that can remain effective even when attackers employ private mempool services [34]. To address these challenges, researchers have proposed shifting focus from detecting adversarial transactions to identifying adversarial contracts. This approach, exemplified by the LookAhead system, leverages common attack patterns, code semantics, and intrinsic characteristics found in malicious smart contracts. By analyzing contracts rather than transactions, this method aims to proactively identify potential attack intentions and victims, even in cases where private adversarial transactions are used [34]. Empirical observations reveal that adversarial contracts often share similar patterns. For instance, over 70% of contracts in a studied adversarial contract dataset used funds from anonymous sources for deployment, and more than 98% were closed-source. Additionally, specific attack types, such as flashloan and reentrancy attacks, require the implementation of complex function call chains and attack logic within callback functions. These patterns and behaviors can be leveraged to improve the identification of potentially malicious contracts, offering a new direction in the ongoing effort to secure the DeFi ecosystem against sophisticated attacks [34].

2.3 Traditional vulnerability detection tools

2.3.1 Code Auditing

Software development encompasses a range of software quality assurance activities, including defect management, testing, and code review [35,36]. In the context of code review, reviewers evaluate whether the source code meets both functional and non-functional requirements. On the other hand, blockchain-based projects also require a rigorous quality assurance process to ensure the quality, security, and overall reliability of the codebase. This process is called code audit. Code auditing and code review are both critical processes in software development. While they share similarities, they differ in scope, purpose, and complexity [8]. In the context of smart contracts, code auditing can be a complex process that involves a comprehensive and in-depth analysis of the whole codebase of the project with the objective of identifying vulnerabilities, potential attack vectors, and security weaknesses. This process involves different tasks, including fault localization, issue explanation, documentation, and issue fix suggestions, which can be time-consuming and highly dependent on the qualifications of the auditors.

2.3.2 Static analysis based approaches for smart contracts auditing

Static analysis tools have become crucial instruments for detecting vulnerabilities in smart contracts prior to deployment. These tools employ various techniques such as symbolic execution, pattern matching, and dependency graph analysis to identify potential security flaws in smart contract code. Popular tools Mythril [37], Manticore [38], Slither [39] and Oyente [40] have been widely adopted in the Ethereum ecosystem, each offering unique approaches to vulnerability detection [41]. For instance, Oyente uses symbolic execution to analyze control flow graphs, while Slither employs Static Single Assignment (SSA) for its analysis. These tools aim to detect a range of vulnerabilities, including reentrancy attacks, exception handling issues, and transaction ordering dependencies.

Despite their widespread use, static analysis tools face several challenges in effectively detecting vulnerabilities in smart contracts. One significant issue is the high rate of false positives and false negatives, which can lead to overlooking critical vulnerabilities or wasting resources on non-existent issues. A study by Ghaleb and Pattabiraman [42] revealed that even well-known tools failed to detect a significant number of injected vulnerabilities, with undetected bugs ranging from 129 to 4137 across seven different bug types. Additionally, these tools often struggle with context understanding, failing to capture the full scope of smart contract interactions, which is crucial for identifying complex vulnerabilities that may arise from inter-contract dependencies [43].

To address these limitations, researchers are exploring advanced approaches that combine traditional static analysis with other techniques. For example, Chen et al. proposed a method that integrates static analysis with a multi-objective optimization algorithm to improve vulnerability detection accuracy and efficiency [44]. Other innovations include the use of machine learning models to enhance semantic understanding of smart contract code and the development of cross-contract analysis techniques to better capture inter-contract vulnerabilities [45]. These advancements aim to improve the precision and recall of vulnerability detection while addressing the scalability challenges posed by increasingly complex smart contracts.

2.3.3 Dynamic analysis and machine learning-based approaches for smart contracts auditing

Dynamic analysis, on the other hand, entails observing the contract's behavior during its execution within a controlled environment. The objective is to unearth vulnerabilities that might not be evident in a static examination but manifest during run-time. Instruments like

Ganache and Truffle Suite are frequently employed for this purpose in the Solidity ecosystem. By simulating transactions and monitoring contract interactions in real-time, dynamic analysis can spotlight issues like gas inefficiencies, unpredictable state changes, and faulty event emissions, which might evade detection during static scrutiny. One of the challenges of using dynamic analysis is that it can be time-consuming and resource-intensive.

Another approach involves using formal verification and symbolic execution techniques to mathematically prove the correctness of the smart contract. Formal verification uses mathematical models to check the correctness of the code, which can be used to identify logical errors and ensure that the smart contract behaves as intended. Symbolic execution, a key technique in this domain, systematically explores multiple execution paths of a smart contract by treating inputs as symbolic values. This allows for a more comprehensive analysis of the contract’s behavior under various conditions. For instance, the Manticore framework, as described by Mossberg et al. [38], employs symbolic execution to analyze both traditional binaries and Ethereum smart contracts, offering a flexible architecture that supports diverse execution environments. Recent advancements in smart contract analysis have seen the integration of machine learning and natural language processing techniques with traditional static and dynamic analysis methods. For example, Liu et al. proposed SmartGift [46], a framework that leverages graph neural networks (GNNs) and natural language processing (NLP) to detect vulnerabilities in smart contracts. This approach combines code semantics with textual information from comments and function names to enhance vulnerability detection accuracy. These hybrid approaches aim to overcome the limitations of traditional methods by incorporating more contextual information and learning from patterns in large datasets of smart contracts.

2.4 LLM-based approaches for smart contracts auditing

Recently, the application of machine learning and deep learning techniques to analyze Solidity smart contracts has gained significant traction [11, 12]. Deep learning models, especially Recurrent Neural Networks (RNNs) and their variant, Long Short-Term Memory (LSTM) networks, have been employed to capture the sequential nature of code and its inherent patterns [12]. LSTMs, with their ability to remember long-term dependencies, are particularly suited to recognizing intricate patterns in Solidity code that might be indicative of vulnerabilities. While this approach has shown outstanding results, its generalization to unseen smart contracts written in a new version of Solidity remains a concern. Also, these models are not able to generate human-like explanations of the found vulnerabilities and the mitigation steps.

On the other hand, the Generative Pre-trained Transformer (GPT) has attracted a lot of attention from the research community regarding its ability to review and generate code [13–17]. Particularly in [18], a comprehensive evaluation of GPT-3.5 and GPT-4 code interpreter for vulnerability detection in smart contracts is illustrated. This work suggests using prompt engineering for the auditing of smart contracts. In particular, it first uses GPT to generate the prompt that describes the task of auditing; then, this prompt is used along with the full source code of the contracts as input to GPT. To measure the performance of GPT, the work considered only the Precision, Recall, and F-measure metrics. However, no qualitative evaluation of the generated output was made. The work did not consider the non-deterministic nature of the GPT model regarding the prompt design. In addition, this work did not evaluate GPT’s capacity to explain the detected vulnerabilities and the mitigation steps. The experiments showed that the prompt length, i.e., the contract length, can negatively affect the detection performance when exceeding the token limit.

Similarly, in [19] a quantitative evaluation of the performance of GPT-4 code interpreter for the task of vulnerability detection has been performed. In particular, the prompt design included a description of a specific attack type besides the whole source code of the contract. Common decentralized finance attacks have been classified into 38 Common Weakness Enumeration (CWE), and each smart contract has been prompted along with a description of the specific attack. Although the results of the experiment have shown great potential for GPT-4 code interpreter to detect vulnerabilities, this experiment has shown that GPT’s performance is inversely proportional to contract length. The authors in [16] suggest combining fine-tuning and LLM-based agents to detect and explain vulnerabilities in smart contracts. It employs a multi-step process where the model first identifies potential issues, then reasons about the underlying causes, and finally ranks and critiques these explanations to ensure the most accurate and justified results.

The authors in [21] present an innovative framework that utilizes a multi-agent conversational approach for detecting and analyzing vulnerabilities in smart contracts. This approach incorporates two primary strategies: Broad Analysis and Targeted Analysis. Although the evaluation demonstrates enhanced effectiveness in identifying vulnerabilities, the tool is highly resource-intensive.

Additionally, its performance can degrade when analyzing longer code segments. In [22], authors suggest an approach leveraging fine-tuned GPT-4 to identify and fix vulnerabilities in Solidity code by creating an annotated Control Flow Graph (CFG)

2.5 Automated test case generation for smart contracts

The proliferation of smart contracts has established them as a cornerstone of automation and trust, facilitating secure and transparent transactions across diverse industries, including finance, real estate, and supply chain management [1]. However, their increasing adoption unveiled many challenges. The immutable and transparent nature of smart contracts paradoxically renders them susceptible to exploitation, misuse, and unforeseen vulnerabilities. Consequently, the imperative to conduct rigorous audits of smart contracts arises, entailing a meticulous examination of their code and features to detect and mitigate potential risks. The immutable nature of smart contracts adds an extra layer of importance for developers and increase the need of the development of a comprehensive and complete test suite [47].

Over the years, various tools have been developed to automate the task of test case generation for smart contracts. Numerous studies have explored the use of genetic algorithms for test case generation [48, 49]. For example, the authors of [50] investigated the application of meta-heuristic algorithms, implementing multiple search techniques such as random search and genetic algorithms (e.g., NSGAII [50], MOSA [50], and DynaMOSA [51]) for effective test case generation. While these approaches have demonstrated promising results, the genetic algorithm approach remains computationally expensive.

However, as demonstrated by the in-depth study conducted by Zou et al. [52], generating test suites for smart contracts is a complex task, with several challenges emerging throughout their development and implementation. 54.7% of the investigated developer reported the lack of mature and business-ready tools for blockchain-specific development. Furthermore, these tools do not consider all corner cases and scenarios, which is the most critical challenge raised by the interviewed developers.

The emergence of large language models (LLMs) for software development tasks [36] has sparked interest in leveraging these models for test case generation [53]. Several tools and techniques have been proposed in this area. For example, Arghavan Moradi Dakhela et al. [54] suggested using mutation testing to guide LLMs in generating effective test cases. After an initial mutation campaign, their approach prompts the LLM with surviving mutants, requesting the generation of tests to kill the mutants. While their empirical evaluation showed promising results, the approach has yet to be tested on large datasets and real-world projects. Large codebases introduce three significant challenges: (1) the large number of generated mutants makes mutation testing computationally expensive and time-consuming, (2) a large number of tests is required to achieve a high mutation score, and (3) real-world projects require intricate test setups and environments and more complex test case than

simple assert-statement tests.

2.6 Chapter Summary

In this section, we conducted an in-depth examination of state-of-the-art vulnerability detection tools and techniques employed in smart contract security. This review encompasses static analysis, dynamic analysis, and machine learning-based methods, each of which plays a critical role in identifying security vulnerabilities within smart contracts. We critically analyzed the strengths, limitations, and applications of these approaches, highlighting how each technique contributes to the current landscape of smart contract auditing.

Additionally, we reviewed cutting-edge methods that leverage large language models (LLMs) for the automated auditing of smart contracts. This includes an analysis of prompt engineering techniques and model fine-tuning strategies aimed at optimizing LLMs for vulnerability detection in smart contracts. The purpose of this review is to establish a foundational understanding of the methodologies and concepts central to the studies and contributions presented in this thesis, facilitating a deeper comprehension of the challenges and innovations in automated smart contract auditing.

CHAPTER 3 A CONTEXT-DRIVEN APPROACH FOR CO-AUDITING SMART CONTRACTS USING GPT-4 CODE INTERPRETER

Smart contracts have become the foundation of automation and trust through secure and transparent transactions [1]. They are adopted in various industries, from finance to real estate to supply chain management, promising efficiency, transparency, and trust in a trustless environment [2, 3]. However, the growth in adoption of smart contracts brings along challenges. Smart contracts are vulnerable to exploits, misuse, and unforeseen vulnerabilities due to the very features that make them powerful (such as immutability and transparency) [4]. This underscores the necessity to audit smart contracts, a process that examines the code and features of the contracts to identify and mitigate potential risks [8].

Smart contract auditing is typically performed manually by specialized auditing engineers and requires a deep understanding of the business use case and the historical attack events [10]. Auditing smart contracts is time-consuming, challenging and dependent on the auditor’s expertise and experience [10]. Auditors require a diverse skill set that encompasses proficiency in debugging, a deep understanding of various code analysis techniques, knowledge of blockchain platforms (e.g., EVM), and a grasp of the business domain and rules embedded within the protocol’s code [10].

A possible solution to address these challenges is using Large Language Models (LLMs) for smart contract auditing. LLMs, empowered by their vast pre-trained knowledge, have demonstrated remarkable capabilities in understanding and generating code [55]. Recently, different studies focused on exploring the usage of LLMs for automating code review, including smart contracts [56]. Nevertheless, LLMs can be susceptible to the phrasing and length of the context in the input prompts. Subtle phrasing, tone, or context changes of a prompt can lead to significantly different model output, especially when the project’s source code is very long and the dependencies are very complex. As shown in a recent study by David et al. [19], increasing the length of the prompt context reduced the performance of the LLM in detecting vulnerabilities. Similarly, Chen et al. [18] suggest that GPT models may encounter detection failures in long smart contract code due to inherent token limitations. Therefore, an appropriate prompt design approach taking into account context phrasing and length is necessary in order to assess LLMs for detecting vulnerabilities and generating appropriate audit reports.

In this chapter, we introduced a novel context-driven prompting technique for smart contract auditing that aims to provide appropriate wording and enough code context based on direct

code dependencies to uncover vulnerabilities and provide code audit recommendations. To guide the LLM in identifying the vulnerabilities, we used the Common Weakness Enumeration (CWE) [19] to reduce the search space, in addition to a set of predefined Common Audit Questions (CAQ) extracted from previous audit reports to scope the generated reports. Moreover, to provide sufficient code context while avoiding including the entire code within the prompt, we developed a chunking approach based on code call graphs, which we refer to as Code Call List or CCL-based chunking. For evaluation, we conduct experiments to answer the following research questions:

1. **RQ1. How does CCL-based chunking perform in comparison to full code prompting?** The concept of chunking, particularly CCL-based chunking, is inspired by the manual auditing process employed by expert auditors. During the audit of a smart contract, auditors meticulously verify each function to ensure it is free from vulnerabilities. This necessitates a comprehensive understanding of all the code executed within each code snippet. For Research Question 1 (RQ1), we assess the performance of CCL-based chunking relative to full-code prompting. In full-code prompting, the GPT-4 code interpreter typically executes automated mechanical chunking by dividing the code based on the maximum token size permissible for a single prompt. Our evaluation aims to determine the effectiveness of CCL-based chunking in comparison to this automated approach.
2. **RQ2. What is the impact of different prompt phrasing approaches (CWE-based, or CAQ-based) on the effectiveness of the GPT results of CCL-based chunking?** The objective of RQ2 is to study the impact of utilizing different wording and phrasing techniques for context augmentation based on expected results (e.g., the expected vulnerability in CWE-based) or generic auditing inquiries (searching for all vulnerabilities in CAQ-based) approaches on the cost and effectiveness of CCL-based chunking to guide the ChatGPT to retrieve the correct results.
3. **RQ3: To what extent does a human auditor’s analysis of GPT-4 code interpreter’s results on an unlabeled dataset align with our previous findings?** The objective of RQ3 is to evaluate the reliability of our proposed prompting approach by assessing its consistency with manual auditing by expert auditors from our partner (a world-leading smart contract auditing company). A second experiment was conducted using unlabeled datasets, and expert auditors were invited to perform an independent analysis and qualitative assessment of GPT-4 code interpreter’s responses, providing an additional layer of verification to our proposed methodology.

The main contribution of the work is as follows:

1. We introduce a context-driven approach to enhance LLM performance in smart contract auditing by proposing three scoping techniques: Code Scoping, Assessment Scoping, and Reporting Scoping. These techniques aim to improve understanding and generate accurate audit reports by addressing the challenges of (i) long code contexts in prompts, (ii) context related to the audit process, and (iii) context related to expected target responses.
2. For each scoping technique, we provide a practical implementation. Code Scoping is realized through CCL (Code Call List), which organizes code into logical chunks based on function call dependencies to improve contextual awareness. Assessment Scoping is realized through CAQ (Common Audit Questions), a tailored set of questions that guide the LLM’s focus on specific vulnerabilities. Reporting Scoping is performed through CWE (Common Weakness Enumeration), helping the LLM categorize and report vulnerabilities using standardized industry classifications.
3. We present hypotheses regarding the effectiveness of the scoping techniques and empirically test these through experiments to assess their impact on audit quality.
4. We provide a new dataset of vulnerable functions, manually labeled by two professional smart contract auditors, to aid in the evaluation and validation of the proposed methodologies.

3.1 Approach

3.1.1 Motivation example

Co-auditing with Generative AI

Co-auditing refers to the practice of involving multiple auditors, which can include both artificial intelligence (AI) systems and human oversight, in the auditing process. [55]. A practice that witnessed fast advancement with the emergence of LLMs. The need for co-auditing can vary depending on the domain, but it is crucial in applications where factual errors can have serious repercussions [57]. An example of such an area is smart contract auditing. Recently, different studies have focused on exploring the use of LLMs for code auditing [18, 36]. While these works highlight the capacity of such models for Solidity code understanding and interpretation, the accuracy of LLMs for the task of detecting vulnerabilities is still low. In [18],

different limitations have been reported for the use of GPT-4 code interpreter for code audits, including uncertainty and context length. The performance of GPT-4 code interpreter can degrade considerably when the prompted code becomes very long. Unlike the standard GPT-4 model, which focuses solely on language understanding and generation, the GPT-4 Code Interpreter is designed to process code snippets and execute them in a secure, sandboxed environment. These long contextual dependencies can lead to ambiguity in properly understanding the prompted codebase. Besides, code auditing requires knowledge about the whole code base, making it more complex for tools such as GPT.

Despite the theoretical capacity of large language models (LLMs) like GPT-4 to process large number of tokens, the actual context window length is less than the declared limit [58]. Our initial experiments reveal that their efficacy in smart contract vulnerability detection is hindered by limitations in contextual understanding, particularly when encountering complex code structures. Notably, the performance degradation of LLMs in this domain is not solely attributable to context length, but rather a multifaceted issue encompassing the model’s ability to capture nuanced code relationships, reason about indirect dependencies, and generalize to novel contract architectures. The manual tests we performed revealed that GPT-4 code interpreter fails to detect all the vulnerabilities in a smart contract when the codebase is too long, inducing many false negatives. This underscores the need for innovative prompting techniques that can effectively harness the strengths of LLMs while mitigating their limitations in smart contract auditing. The integration of generative AI in code auditing poses a unique challenge, as highlighted by Liu et al. in "What It Wants Me To Say" [59]. The disconnect between the end-user programmer’s requirements and the output of large language models (LLMs) can lead to suboptimal results in tasks like code generation and vulnerability detection. In the context of code auditing, this gap can have significant implications, as LLMs may not fully comprehend the nuanced security concerns and auditing standards that human auditors take for granted. However, LLMs can still play a vital role in augmenting the code auditing process, particularly in tasks such as:

- Identifying potential vulnerabilities and anomalies in large codebases
- Providing recommendations for code improvements and security enhancements

The authors emphasized the role of scoping, elaboration, language restructuring, and intent shaping in grounded utterance [59]. Stemming from these results, this work focuses on approaches to improving co-auditing through context scoping and augmentation. We particularly focus on the following three scopes:

1. **Code scoping:** This scoping is related to managing long code with complex inter-dependencies. In such cases, scoping should include methods to direct the LLM’s attention to the interrelated code elements that could be far apart but have close interactions. Understanding how code elements interact could lead to identifying potential vulnerabilities.
2. **Reporting scoping:** This scoping is related to the output audit report; an auditor needs to report specific content in a specific order and format. This is highly dependent on the auditing process. Scoping this information through specific questions not only guides the LLM to generate the required information but also focuses attention on the next tokens that need to be generated, which could positively influence the output results.
3. **Assessment scoping:** This scoping is related to defining the specific vulnerabilities, weaknesses, or security issues an auditor is targeting within the codebase.

Researchers have been investigating various aspects of context in large language models (LLMs), including how it is defined, utilized, and evaluated [58, 60]. In [58], authors address the challenge of long-context understanding for LLMs. The chapter defines the context window as the maximum amount of text or tokens that a model can process in a single input. The study revealed that LLMs excel in short dependency tasks but struggle with more complex long dependency tasks, indicating a gap in true long-context understanding. Yilun Zhu et al. [61] conducted an empirical investigation into the contextual comprehension capabilities of LLMs. Their systematic evaluation assessed the LLMs’ abilities to accurately interpret subtle and implicit contextual information, a crucial factor in tasks requiring profound contextual awareness. The findings revealed that while LLMs exhibit satisfactory performance in general linguistic tasks, they frequently misinterpret or fail to comprehend nuanced contextual cues. This study underscores the significance of contextual definition and structure in optimizing LLM performance, particularly for code generation and vulnerability detection tasks. Notably, LLM performance degrades when processing large codebases, characterized by intricate dependencies and numerous function calls, even when token counts remain within contextual windows [62]. For comprehensive codebase understanding, semantic relationships between disparate code segments assume paramount importance in structuring input prompts. Harnessing the structural properties of function call graphs, throw code scoping, offers a promising approach to represent logical connections between functions, thereby enhancing contextual awareness.

To inform the development of effective LLM auditing queries for smart contracts, we employed

a mixed-methods approach combining card sorting methodology and expert elicitation. Our investigation involved analyzing a dataset of audit reports extracted from Code4Arena, a prominent capture-the-flag platform utilized by the auditing community, to identify key themes, patterns, and vulnerabilities prevalent in smart contract audits. In collaboration with professional auditors, we distilled these findings into a comprehensive set of tailored questions designed to elicit relevant information from the LLM, ensuring alignment with industry auditing standards. By integrating human expertise and real-world audit report data, our approach establishes a robust foundation to explore the potential of LLMs in automated smart contract auditing, providing a systematic and informed basis to enhance the efficiency and effectiveness of this critical process.

This study integrates LLMs into real-world smart contract auditing workflows, enhancing auditing practice by tailoring LLM outputs to specific vulnerability detection and reporting requirements. By bridging the gap between LLM general capabilities and smart contract security needs, our approach addresses specifically auditing smart contracts. Our focused methodology provides actionable insights into LLM applicability, rather than broadly investigating LLM behavior.

In the evaluation section, we provide an example of implementation of our understanding of the context definition for the task of auditing smart contracts.

Motivation Example

Listing 3.1 illustrates an example of a smart contract called TokenizingVault. This vault contract takes users' diverse ERC20s (token standard) deposits and then issues corresponding "credits" as deposit tokens. These tokens can then be exchanged to reclaim the initial deposit. This design provides fungibility that allows users to trade their secured assets seamlessly. The TokenizingVault contract imports two other contracts, namely ERC20.sol and ERC20CreditToken.sol. ERC20 and ERC20CreditToken also import different other contracts which we tried to prompt this contract along with the other imported contract. The contracts source code length exceeds the 8000 token limit supported by GPT-4. A direct solution for such a challenge is a naive truncation. We prompted the source code of the contracts sequentially in different prompts. We used a binary classification of the contract and asked GPT to locate the vulnerable functions, if any, and explain the vulnerability.

Listing 3.1 TokenizingVault contract

```
pragma solidity 0.8.15;
```

```

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {ERC20CreditToken} from "./ERC20CreditToken.sol";

contract TokenizingVault {
    /// ERC20 accounting tokens (unique per underlying)
    mapping(ERC20 => ERC20CreditToken) public creditTokens;
    /// ERC20CreditToken reference implementation
    ERC20CreditToken public immutable creditTokenImpl;
    constructor() {
        creditTokenImpl = new ERC20CreditToken();
    }
    function create(ERC20 underlying, uint256 amount)
        external nonReentrant returns (ERC20CreditToken, uint256)
    {
        ERC20CreditToken creditToken = creditTokens[underlying];

        // Revert if no token exists, must call deploy first
        if (creditToken == ERC20CreditToken(address(0x00)))
            revert('Token Does Not Exist');

        // Transfer in underlying
        underlying.transferFrom(msg.sender, address(this), amount);

        // Mint new credit tokens
        creditToken.mint(msg.sender, amount);

        return (creditToken, amount);
    }
    function redeem(ERC20CreditToken token, uint256 amount)
        external nonReentrant
    {
        token.burn(msg.sender, amount);
        token.underlying().transfer(msg.sender, amount);
    }
    function deploy(ERC20 underlying)
        external nonReentrant returns (ERC20CreditToken)
    {
        // Create credit token if one doesn't already exist
        ERC20CreditToken creditToken = creditTokens[underlying];
        if (creditToken == ERC20CreditToken(address(0))) {
            bytes memory tokenData = abi.encodePacked(underlying, address(←
                this));
            creditToken = ERC20CreditToken(address(creditTokenImpl).clone(←
                tokenData));
        }
    }
}

```

```

        creditTokens[underlying] = creditToken;
    }
    return creditToken;
}
}

```

Due to the complex dependency, the code base is prompted using a sequence of 5 prompts. GPT is composed of 3 parts: general observations, Potential Vulnerabilities and Suggestions, and conclusions. As we can see in the generated response¹, GPT fails to detect a crucial vulnerability which is related to the *redeem* function.

The core issue for this contract is the function *redeem*. The function *redeem* is external and does not have an access control list or input validation schema. This function is vulnerable because it does not validate if the passed token is authenticated. One exploitation scenario is that an attacker could invoke the *redeem* function using a maliciously crafted ERC20CreditToken. If this token's *underlying* method points to a legitimate and valuable ERC20 token that other users have stored in the vault, it may lead to unauthorized withdrawals. The vulnerable function is only three lines long. Although GPT's response mentioned different code issues related to the contract, the vulnerability has not been listed. We repeated the experiment various times. The potential reasons can be:

- **Lack of context depth:** In the absence of specific context description, an LLM is capable of generating outputs in formats that are not uniformly structured. This irregularity in the generated response presents substantial obstacles in the extraction of valuable insights that are both accurate and coherent. The absence of a precise description of the desired output in terms of the context inquiry process, related to the questions that need to be answered, and in terms of the generated response, related to expected results template, can lead GPT to generate high-level results that can increase the rate of false positive/negative and/or missing certain complex vulnerabilities.
- **Sensitivity to source code length:** The length of the prompt context and the code can affect the performance of LLMs for vulnerability detection [18, 19], where the effective maximum token number for GPT-4 is lower than 8000.

This suggests that a more effective chunking technique for the source code is required to fully exploit the LLMs code understanding capabilities. Hence, this work proposes a new chunking technique based on a function call graph for source code. Moreover, the chapter will present

¹<https://chat.openai.com/share/fe0be4f0def-449a-bf22-30fd9cc373cb>

comparative experiments involving different groups of prompt engineering techniques for context representation and assess the generated audits, demonstrating the current advantages and limitations of context length on the performance of LLMs for smart contract audits.

3.1.2 Approach

In this section, we propose three context-augmenting approaches with the goal of addressing the problems of (i) long code contexts in a prompt, (ii) context related to the process of inquiry (i.e., to augment knowledge related to the audit process extracted from previous audit reports), and (iii) context related to the expected target responses (i.e., inquiries about a specific issue or weakness based on the knowledge from previous vulnerabilities). These augmentation approaches can be used to design prompts that can support smart contract co-auditing with the support of GPT. Figure 3.1 illustrates a high-level overview of our approach. The rest of this section explains the three context-augmenting approaches and their corresponding prompt templates.

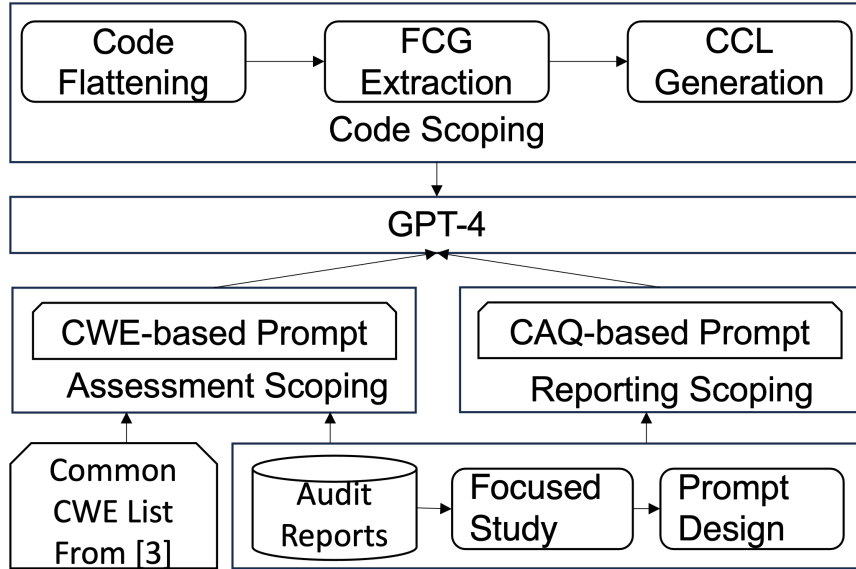


Figure 3.1 Overview of the study method

Code Scoping

To provide sufficient code context while avoiding including the entire code within the prompt, we propose a chunking approach based on Code Call List (CCL). This process includes code flattening, Function Call Graph (FCG) extraction, and finally, CCL generation.

- **Code Flattening**

Smart contract code often involves importing, calling, and referencing other contracts and libraries and implementing interfaces that extend the contract’s functionality. These dependencies can increase the complexity of auditing smart contracts, especially when dealing with multiple imported contracts. To simplify this complexity, we employ a code-flattening approach, which consolidates all the code from the contract and its dependencies into a single file. The flattening process consists of combining all the solidity files, dependencies, imported libraries, and merging them into one continuous file. This includes parsing the solidity files, resolve imports and dependencies to make sure that all the code is included in the flattened file. In the flattening process, we identify and include the functions within the main contract and the call relationships between them, including their interactions with interdependent functions from imported contracts. Comments are retained because they offer valuable context for understanding the intended functionality of the code. Moreover, the process ignores unreachable code for simplicity. Flattening a contract creates a single file that includes all code from the contract and its dependencies, eliminating the need to manage multiple files and ensuring that all dependencies are included correctly. To achieve this code flattening process, we utilized the Slither contract flattening tool [39], specifically employing the OneFile strategy. Slither is a static analysis tool that detects potential issues in Solidity code, including problems related to imports and interfaces. The OneFile strategy merges all contracts within the project into a single, standalone file. By using Slither’s OneFile strategy, we streamline the codebase and create a consolidated, easily accessible code structure for efficient analysis and development.

- **Function Call Graph (FCG) Extraction**

The single file resulting from the flattening process is long and cannot fit within one prompt. One way to provide GPT with enough context is to focus on the FCG, which shows the flow of function calls, indicating which functions are called by other functions and how they are connected. FCG is utilized in our co-auditing approach to aid the segmentation mechanism to send only logically related code segments based on interdependencies rather than their physical location within the codebase. In our work, to generate a FCG from the flattened solidity file, we again used Slither to extract function calls from each contract. Slither uses the context of function calls and applies static and data flow analysis techniques to distinguish between internal and external function calls. The extracted function calls are then consolidated into a call graph. Based on the smart contract code in Section 3.1.1, the generated FCG is shown in figure 3.2.

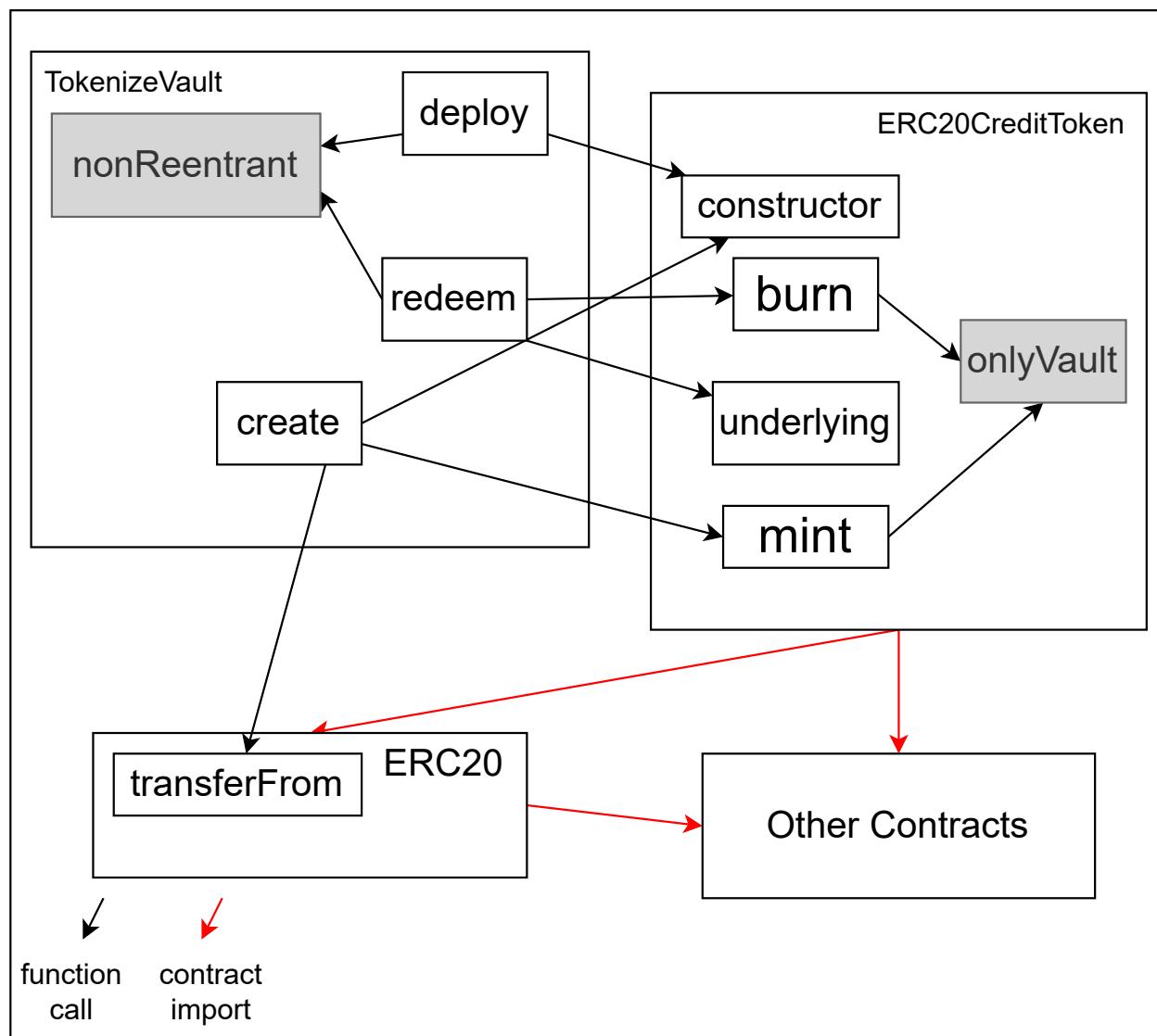


Figure 3.2 Function call graph of the `tokenizeVault` contract

- CCL Generation

Algorithm 1 CCL generation

Input: flattenedContract, adjacencyList

Output: codeCallList

```

1: Initialize codeCallList as an empty dictionary
2: for each functionId, callRelationships in adjacencyList do
3:   for each callRelationship in callRelationships do
4:     callerFunctionId, calledFunctionId  $\leftarrow$  callRelationship
5:     functionCode  $\leftarrow$  extractCode(calledFunction, flattenedContract)
6:     codeCallList[functionId].append(functionCode)
7:   end for
8: end for
9: return codeCallList

```

Based on the FCG, we generate the CCL for each function. A CCL is a list containing all the function codes that are called during a specific function execution, which provides the GPT-4 code interpreter with a self-containing list of function codes. The number of CCLs corresponds to the number of functions implemented within the smart contract. Each CCL is composed of the code of the audited function, the caller function, and the code of the callee functions. This process is also recursive for the callee functions. This process guarantees that all the code being executed during a specific function call is included within the prompt. Our approach differentiates between external and internal function calls in the contracts. Using the function name and the type of the call, we locate and extract the exact called function and its corresponding code. This aims to provide the LLM with enough context to fully process the functions and recognize the vulnerability patterns. The listing 3.2 illustrates a sample of CCL generated from the FCG for the function *deploy*.

Listing 3.2 CCL list of the function 'deploy' extracted from contract TokenizingVault

```

['function deploy( ERC20 underlying ) external nonReentrant returns (←
  ERC20CreditToken) { // Create credit token if one doesnt already ←
  exist ERC20CreditToken creditToken = creditTokens[underlying]; if (←
  creditToken == ERC20CreditToken(address(0))) { bytes memory ←
  tokenData = abi.encodePacked( underlying, address(this) ); ←
  creditToken = ERC20CreditToken( address(creditTokenImpl).clone(←
  tokenData) ); creditTokens[underlying] = creditToken; } return ←
  creditToken; }',

```

```

'constructor(string memory name, string memory symbol, ERC20 ←
    _underlying) ERC20(name, symbol) { underlying = _underlying; vault = ←
    msg.sender; }',
'modifier nonReentrant() { _guardCounter += 1; uint256 localCounter = ←
    _guardCounter; _; require(localCounter == _guardCounter); }']

```

Reporting Scoping - CAQ Prompt Design

The effectiveness of LLMs in audit reporting heavily relies on how specifically the questions are posed. Accurate and relevant responses from LLMs depend on question precision, which is crucial in audit reporting, where delivering exact content in a certain order and format is essential. By formulating targeted questions, we direct the LLM to produce the required content accurately and focus on generating subsequent relevant information. Therefore, integrating a response template into the prompt is a key strategy to reduce response variability and standardize the output format of LLMs in audit reporting. In order to do this, we adopted a focus group methodology to identify the essential components of audit reports². Initially, a diverse collection of existing audit reports from our partner and from Code4rena³ is assembled for analysis to understand the structure and content of audit reports. Then, a focused group of two professional auditors, in addition to three students, examined the reports. Each member of the group was asked to extract a schema of the main report parts. The group then agreed on a common schema that represents key report components that need to be included in the audit report. Based on the discussion, the group agreed that a report needs to specify (i) if a vulnerability exists, (ii) a proof of the concept of the vulnerability, (iii) the business impact of the vulnerability, and (iv) suggestions for potential fixes. Building on this, we create the CAQ-based prompt strategy, then refine it to address Assessment Scoping, as we explain in the next section. The CAQ-based prompt strategy aims at eliciting specific, relevant responses in alignment with these identified components. Pilot testing and iterative refinement based on feedback from both the professional auditors and the students were conducted on a small dataset, leading to a finalized template that generated required, human-like responses. To limit the impact of the non-deterministic nature of generative models, we fixed the temperature in all tests to 0 in the GPT-code-interpreter during the pilot testing. Figure 3.3 illustrates the CAQ-based prompt structure. To get a complete audit report based on this approach, the number of prompt executions is equal to the number of functions in a smart contract.

²<https://code4rena.com/reports/2024-03-dittoeth>

³<https://code4rena.com/reports>

Analyze the following code, respond me in this format (fill in the part):

1. Are there any vulnerabilities? Provide a brief initial response with one of the following: 'Yes,' 'No,' or 'Not sure.' If your answer is 'No,' you can stop here. If your answer is 'Yes,' proceed to the next steps.
2. Explain in details how this vulnerability can be exploited :__ __ __.
3. The business impact of this vulnerabilities in one sentence: function leads to.
- 4.The potential solutions of this vulnerabilities.

Note: You can state more than one lines which have this vulnerability! This is a short extraction of the solidity code and please assume all the variables in this code have been defined.

Figure 3.3 CAQ-based prompt

Assessment Scoping - CWE Prompt Design

To incorporate context related to the expected target responses, we refined the reporting template by incorporating information about the common expected vulnerabilities and weaknesses. To do so, we followed a similar approach to the work in [19]. In this work, the authors analyzed a set of 52 attacked Decentralized Finance protocols that have led to 952M USD in loss. The exploited vulnerabilities in the smart contracts have been categorized into 38 categories. In this chapter, we use this list of the 38 most common vulnerability types to refine the prompt design. The aim is to make it vulnerability-specific. The LLM is prompted sequentially to identify only one type of vulnerability at a time from the CWE list. In the prompt, we request the LLM to respond with "yes", "no", or "not sure" to a **specific type** of vulnerability. This aims to enhance the guide for the attention mechanism of LLM.

Figure 3.4 illustrates the structure for CWE-based prompts. For each function, we execute this prompt for the 38 vulnerabilities identified in the CWE. In order to generate a complete audit report for a smart contract, we repeat the execution of the prompt for every function in the contract. The total prompt execution required equals 38 multiplied by the number of functions in the contract.

3.2 Evaluation

Chapter Overview. This section evaluates our proposed methods using an empirical study, detailed in Figure 3.5, which outlines our approach to addressing the research questions.

Analyze the following code, respond to me in this format (fill in the part with __ __ __):

1. Are there any [**CWE type**] vulnerabilities? Provide a brief initial response with one of the following: ‘Yes,’ ‘No,’ or ‘Not sure.’ If your answer is ‘No,’ you can stop here. If your answer is ‘Yes,’ proceed to the next steps.
2. Explain in details how this vulnerability can be exploited :__ __ __.
3. The business impact of these vulnerabilities in one sentence: __ __ __
4. The potential solutions of this vulnerabilities: __ __ __

Note: You can state more than one lines which have this vulnerability! This is a short extraction of the solidity code and please assume all the variables in this code have been defined.

Figure 3.4 CWE-based prompt

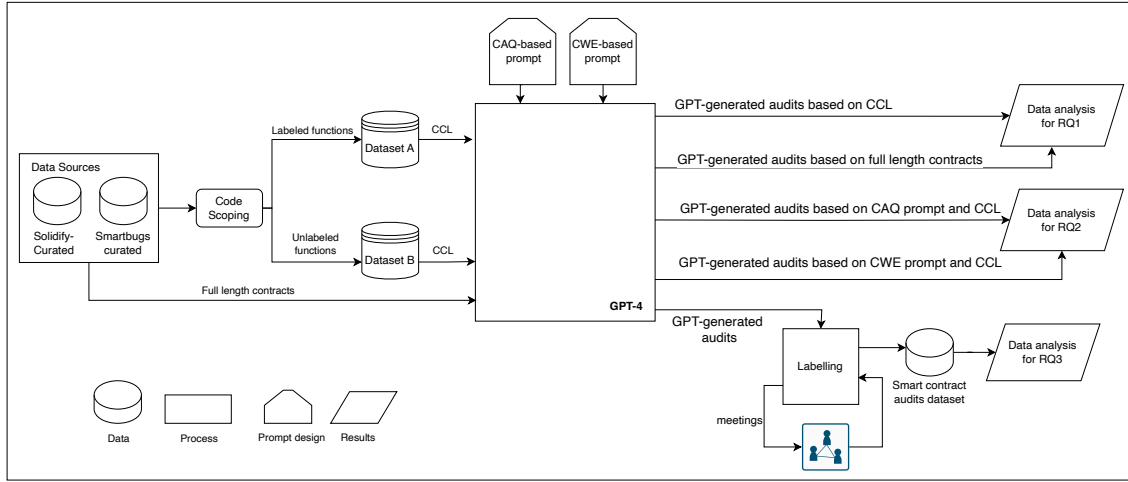


Figure 3.5 Overview of the empirical study approach

3.2.1 Data Sources

Our study utilizes two labeled datasets: Smart Bugs Curated [41] and SolidiFI-Benchmark [42]. Smart Bugs Curated is an academic collection of 69 vulnerable smart contracts that are categorized into ten categories based on vulnerability types, such as access control, arithmetic, and bad randomness [41]. Each contract is annotated with the precise location and classification of its vulnerabilities. The SolidiFI-benchmark repository contains 50 vulnerable contracts covering seven bug types, including reentrancy, timestamp dependency, unhandled exceptions, and unchecked send. Each folder represents a specific vulnerability type. The repository also includes injection logs that identify the exact locations and types of bugs introduced in the code.

The vulnerable smart contracts in both datasets are classified into different categories. Each

smart contract belongs to only one category, and only one function of the smart contract is vulnerable. During our preliminary study, we noticed that GPT revealed different vulnerabilities in one smart contract. These vulnerabilities affect functions that are not claimed to be vulnerable. Moreover, although these datasets contain the labeled vulnerabilities, they might also include other vulnerable functions that are not labeled. To ensure the integrity of the study and given that GPT could have seen the publicly available labeled datasets, we considered unlabeled functions in the datasets. Hence, in this study, we divide the data sources into two datasets: **Dataset A** and **Dataset B**. Dataset A includes the labeled vulnerable functions, while Dataset B has unlabeled functions extracted from the data sources.

3.2.2 Experiment Setup

Table 3.1 Experiment Sources

Sources	Data	Prompt	source code
source 1	Dataset A	CAQ prompt	full code
source 2	Dataset A	CWE prompt	full code
source 3	Dataset A	CAQ prompt	CCL
source 4	Dataset A	CWE prompt	CCL
source 5	Dataset B	CAQ prompt	CCL
source 6	Dataset B	CAQ prompt	full code

As mentioned in section 3.1.2, our approach is based on three scoping strategies: *code scoping* aims to shorten the prompted code, *reporting scoping*, provides GPT with a standardized output format for the audits, and *assessment scoping*, sets a focus at the targeted vulnerability that GPT should address. These strategies were used to guide the design of different prompting templates. In our empirical study, we study the impact of employing these strategies on the GPT performance in smart contract code auditing.

We designed six experiments to answer the research questions based on the different prompting templates and whether they are used with the full code or the Code Call List (CCL). Table 3.1 shows all the experiments; columns show the source name as it appears in the replication repository, the dataset used, the prompt template used, and whether the experiment is based on the full code of CCL, respectively. These experiments were used to answer the questions listed in section 3. In this study, we utilized GPT-4’s code-interpreter plugin with a fixed temperature of 0 and a token limit of 8000. Due to the absence of a GPT-4 API code-interpreter for code interpretation at that time, we employed Selenium—an open-source framework for automating web applications—to interact with the GPT-4 playground. This

Table 3.2 Experimental results on Dataset A : number of detected vulnerable function for each scoping technique

Vulnerability type	CAQ prompt		CWE prompt		Ground truth
	full contracts	CCL	full contracts	CCL	
access control	11	21	17	21	23
bad randomness	9	26	10	29	30
denial of service	3	6	6	6	6
reentrancy	23	32	23	32	32
arithmetic	12	23	18	23	23
front running	3	6	4	6	6
unchecked low level calls	41	49	41	50	54
time manipulation	4	4	3	4	4
short addresses	1	1	1	1	1
other	2	5	2	5	5
total	109	173	125	177	184

setup automated the process of generating and submitting prompts, as well as collecting responses. All the scripts we used for the experiments can be found in this repository.⁴

3.2.3 Empirical Study Results

Impact of the code scoping on the GPT performance

RQ1. How does CCL-based chunking perform in comparison to full code prompting?

In this research question, we evaluate the performance of the proposed CCL chunking approach and compare it to the full code prompting performance using different prompt designs. Consequently, we have divided RQ1 into two more specific questions:

- RQ1.1: When utilizing the CAQ prompt, how does the performance of CCL-based chunking compare to that of full code prompting?
- RQ1.2: When utilizing the CWE prompt, how does the performance of CCL-based chunking compare to that of full code prompting?

⁴<https://zenodo.org/records/10530868>

After prompting GPT-4 interpreter using the different strategies, the responses are collected in CSV files and manually compared to the ground truth. A response is considered correct if GPT correctly identifies the same type and exact location of the vulnerability as described in the ground truth contracts. For each type of vulnerability, we report the number of correctly detected vulnerabilities as shown in Table 3.2.

To answer RQ1.1, we conducted hypothesis testing [63] on results from sources 1 and 3 in Table 3.5. We formulated the null hypothesis (H0) and the alternative hypothesis (H1) as follows:

RQ1.1-H0: *When utilizing the CAQ prompt, there is no statistically significant difference in the performance of CCL-based chunking compared to that of full code prompting.*

RQ1.1-H1: *When utilizing the CAQ prompt, there are statistically significant differences in the performance of CCL-based chunking compared to that of full code prompting.*

To compare the efficiency of these strategies (i.e., CAQ prompt with CCL-based chunking vs. CAQ prompt with full code), which operate as classifiers, we use McNemar’s test, which is particularly useful for comparing the performance of two classifiers. The test statistic measurement indicates a value of 62.01 following a chi-squared distribution with one degree of freedom. For a significance level of 0.05, consulting the chi-squared distribution table for one degree of freedom, we identified a critical value of 3.841. Hence, the test statistic is greater than the critical value; we can conclude that there is a significant difference in terms of effectiveness between the full code and CCL-based chunking methods.

We use Cohen’s d to measure the effect size, which quantifies the magnitude of the difference in detection rates between the full code and CCL-based chunking methods. We chose Cohen’s d because it provides a standardized measure that enables us to interpret the practical significance of the observed differences, complementing traditional significance tests. The Cohen’s d value of -0.45 indicates a moderately significant difference in effectiveness between the two approaches. This result, combined with the detection rates of 59.24% and 94.02% for the full code and CCL-based chunking methods, respectively, and a low p-value of approximately 0.000287, supports the rejection of the null hypothesis (RQ1.1-H0). These findings validate the superiority of the CCL-based chunking method when utilizing the CAQ prompt. Therefore, we conclude that when utilizing the CAQ prompt, there are differences in the performance of CCL-based chunking compared to that of full code prompting, with CCL-based chunking performing much better.

To answer RQ1.2, we conducted a hypothesis test on the results from sources 2 and 4 as presented in Table 3.1. We formulated the null hypothesis (H0) and the alternative hypothesis

(H1) as follows:

RQ1.2-H0: *When utilizing the CWE prompt, there is no statistically significant difference in the performance of CCL-based chunking compared to that of full code prompting.*

RQ1.2-H1: *When utilizing the CWE prompt, there are statistically significant differences in the performance of CCL-based chunking compared to that of full code prompting.*

Following a similar method to RQ1.1, we assessed the effectiveness of different strategies—comparing CWE prompts with CCL-based chunking against CWE prompts with full code. To evaluate the performance differences, we employed McNemar’s test. The test statistic formula for McNemar’s test generated a value of 50.01, which significantly surpasses the chi-squared distribution’s critical value of 3.841 for one degree of freedom. This indicates a statistically significant difference in performance between the two approaches, confirmed by a p-value of approximately 0.00117, well below the conventional significance thresholds. After confirming the statistical significance, we proceeded to examine the effect size to understand the practical impact of this difference. The calculated Cohen’s d value of -0.36 indicates that the average number of correctly detected vulnerabilities is lower with full code prompting compared to CCL-based chunking. Specifically, accuracy rates were 67.93% for full code and 96.02% for CCL-based chunking. Additionally, the sensitivity, F1-score, and precision metrics for full code were 76.69%, 80.91%, and 85.62%, respectively, compared to 98.88%, 98.06%, and 97.25% for CCL-based chunking. The significantly higher precision and F1-score achieved by the CCL-based chunking method under the CWE prompt suggest that this approach not only detects more true positives but also reduces the occurrence of false positives. These results clearly demonstrate the superior performance of CCL-based chunking under the CWE prompt.

Combining the conclusion from RQ1.1 and RQ1.2, we found that:

Finding 1: Under both prompt methods, the CCL-based chunking method performs better than the full code prompting.

Impact of the assessment scoping on GPT performance

RQ2. What is the impact of different prompt phrasing approaches (CAQ-based or CWE-based) on the effectiveness of the GPT results of CCL-based chunking?

Building upon the findings from RQ1, we focus on evaluating the CAQ-based and CWE-based

assessment scoping methods, specifically when applied in conjunction with the CCL-based chunking approach. This experiment is designed to test the following hypotheses:

RQ2-H0: *Using CCL-based chunking, there is no difference in the proportion of found vulnerabilities between the CAQ prompt and the CWE prompt.*

RQ2-H1: *Using CCL-based chunking, there are differences in the proportion of found vulnerabilities between the CAQ prompt and the CWE prompt.*

In order to assess and compare the performance of the CAQ and CWE prompts, we calculate McNemar’s test [64], which results in a value equal to 2.25. This value is inferior to the critical value obtained from the chi-squared distribution table for 1 degree of freedom and a 0.05 confidence significance level. Using this metric, we are not able to reject the null hypothesis, RQ2-H0. The statistical analysis yielded a p-value of approximately 0.553, which fails to provide substantial evidence to reject the null hypothesis, indicating no significant difference. The accuracy rates are 94.02% and 96.02%, respectively. the recall value is equal to 94.02% and 96.02%. To calculate the effect size, we first determined the standard deviations for the CAQ and CWE prompts, which were 15.61 and 16.04, respectively. Using these values, the Cohen’s d test yielded an effect size of -0.025, indicating a very small difference between the two approaches in their ability to detect vulnerabilities. This suggests that while both approaches perform similarly in terms of effectiveness, additional metrics are needed to comprehensively evaluate their performance.

To further analyze their capabilities, we calculated sensitivity, F1-score, and precision. The CAQ prompt achieved sensitivity, F1-score, and precision values of 76.69%, 80.91%, and 85.62%, respectively. In contrast, the CWE prompt showed higher sensitivity at 98.88% and F1-score at 98.06%, but a slightly lower precision of 97.25%. These results reveal nuanced differences in the performance of the two approaches, with the CAQ prompt displaying stronger precision, which is crucial to reduce false positives, while the CWE prompt excels in sensitivity, reflecting its ability to detect a broader range of vulnerabilities. This detailed comparison highlights the trade-offs between the two approaches and underscores the importance of selecting metrics aligned with the specific goals of vulnerability detection.

Consequently, we conclude that, when using CCL-based chunking, there is no statistically significant difference in the proportion of discovered vulnerabilities between the CAQ and CWE prompts. We followed a specific methodology for our assessment as detailed in Section III.C. For each function, we utilized the CAQ prompt once, while the CWE prompt was executed for all 38 vulnerabilities identified in the Common Weakness Enumeration (CWE). Notably, any vulnerabilities not defined in CWE would not be detected by the CWE prompt, potentially leading to false negatives. Therefore, we assert that:

Finding 2: In the context of comparing the performance of the CAQ-based prompt and the CWE-based prompt, it is observed that there is a minimal disparity in their efficacy. However, it is worth noting that the CAQ-based prompt demands less effort and can detect a wider range of vulnerability types.

Integrating the insights derived from RQ1 and RQ2, we can conclude that:

Finding 3: The most effective approach for detecting vulnerabilities in Solidity code using the GPT-4 code interpreter, within the context of both code segment and text description prompts, is the combination of CCL-based chunking and the CAQ-based prompt.

We have redone the tests on the redeem function that we explained in the section 2.3, using the CAQ-based prompt along with CCL, GPT-4 was able to detect effectively the vulnerability⁵.

Human qualitative evaluations

RQ3. To what extent does a human auditor analysis of GPT-4 code interpreter’s results on an unlabeled dataset align with our previous findings?

Addressing RQ3, we aim to evaluate the reliability and add an extra layer of validity to the proposed approach. We conducted a qualitative experiment in which we asked expert human auditors to audit and manually validate the auditing reports generated by our approach on a set of unlabeled smart contract functions. This evaluation was conducted under the assumption of a low likelihood that GPT models were previously exposed to, or trained on, the specific function annotations (labels). We hope this will provide insights into how well our approach performs in scenarios likely unfamiliar to the GPT’s training data. The following sections details the dataset, the experiment, and insights from validation.

Data Collection: For this experiment, we used Dataset B, which includes the CCL of 86 functions. These functions were selected from real-world smart contracts. Our selection process involved choosing complex (long) and diverse smart contracts, from which we extracted functions not previously labeled as vulnerable. This selection strategy was based on the presumption that the labels (vulnerabilities) of these unlabeled functions were less likely to be

⁵<https://chat.openai.com/share/fb51d632-c6aa-47b0-a81e-7ec473c28b92>

part of GPT’s training set, ensuring an unbiased and rigorous assessment. The initial results (Finding 1) indicated that CCL-based chunking effectively reduced the size of the prompted source code. However, this leads to an important question: does the public availability of datasets enhance GPT’s ability to detect vulnerabilities?

Dataset B comprises unlabeled functions paired with corresponding Contract Contextual Labels (CCL). To ensure rigor, we selected contracts meeting specific criteria:

- Inter-contract dependencies (importing libraries/contracts): This criterion ensures the dataset captures scenarios where functions interact with external libraries or contracts, reflecting real-world complexity.
- Complex logic and lengthy code (more than 100 lines): This criteria is used to ensure complex contracts functionalities.
- Varied contract lengths (representative sampling): Including contracts of different sizes ensures the dataset covers diverse use cases, providing a balanced evaluation of model performance across both simple and complex scenarios.
- Developer-commented functions (simulating human auditing): Human auditors generally do audits to commented contracts. The comments help auditor to understand the code functionality and thus help them detect logic vulnerabilities.

This diversified dataset challenges Large Language Models (LLMs) and mirrors real-world auditing scenarios.

Annotation and Evaluation

To qualitatively evaluate the results, the auditors manually annotated the GPT-4-code-interpreter’s responses using two criteria; namely, response correctness and relevance:

Response Correctness measures whether the GPT’s judgment is correct regarding the existence of the vulnerability and its location on the code. We noticed that when GPT identifies the type of vulnerability correctly, it also locates the vulnerable location correctly. Correctness is divided into three sub-categories:

- level 1 (NOT) : GPT response is not correct
- level 2 (Partial) : The generated audits successfully detect at least one correct vulnerability.
- level 3 (Perfect) : All the vulnerabilities included in the GPT audit are correct.

Response Relevance measures the quality of explaining how the vulnerability can be exploited. It reflects the clarity of the response and how comparable it is to the human-generated audits. The response relevance is divided into three categories:

- level 1 (NOT) : The response is not relevant compared to a manual audit.
- level 2 (Partial) : The response includes some relevant information that can be helpful to a professional auditor.
- level 3 (Total) : GPT response is similar to manual audits.

We employed a rigorous annotation process for the manual study of GPT-generated audits of the selected 86 CCLs. Two experienced auditors from our industrial partner annotated the samples independently along two dimensions. The annotators have extensive expertise in the development and auditing of smart contract projects. The process of annotating the GPT responses was conducted in rounds. First, each auditor independently reviewed and evaluated the GPT responses based on the correctness and relevance of the responses. Then, a meeting between the two annotators and the co-authors was held to ensure they shared the same understanding of the task. The goal was to enable the annotators to perform their tasks with a well-defined framework and to enhance the quality and consistency of their individual assessments, recognizing that differences in judgment are inherent in this challenging domain. The annotators were able to access the full source code corresponding to each CCL to have an accurate understanding of the functions being audited. Different meetings were held between the auditors to resolve discrepancies between annotations. The final Cohen’s Kappa coefficient [65] for response correctness and response relevance was 0.65 and 0.68, respectively, suggesting moderate agreement between the two annotators. Even though the relevance score between the two annotators is not exceptionally high, the labeling results retain their value. This is because our analysis focuses on comparing the changes in scores assigned by the same annotator to the CCL-based chunking and full code usage performances rather than making a direct comparison between the scores from two annotators. This approach underscores the individual annotator’s consistency and personal benchmark standards, providing a more controlled and internally consistent comparison.

Results: The outcomes of the annotation procedure are depicted in Figure 3.6. The results show that despite using unlabeled datasets, on average, expert auditors concurred that with our prompting approach, the GPT-4 interpreter correctly and accurately generated reports in 56% of the cases, and these reports were relevant in 46% of instances. These percentages increase to 76.5% and 75% for partial correctness and relevance, respectively. This aligns

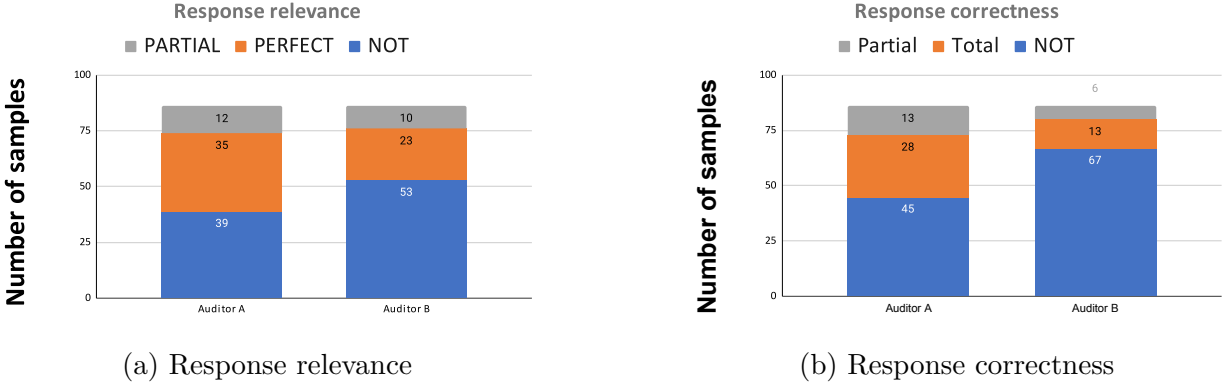


Figure 3.6 Qualitative evaluation of the GPT generated audits of two reviewers

with our prior findings that the application of CCL-based chunking enhances the GPT-4 code interpreter’s ability to identify vulnerabilities in unlabeled datasets. The derived auditing dataset can be accessed openly in the project’s repository.⁶

Finding 4: Analysis from human annotations suggests that, in unlabeled datasets, the use of CCL-based chunking improves the capability of the GPT-4 code interpreter in identifying vulnerabilities.

3.3 Discussions

A. Affirming the Efficacy of Our CCL-Based Chunking Approach

Our CCL-based chunking method, rooted in the function call graph (FCG) framework, is indicative of efficiency in interconnecting related functions at different positions in the code. This approach ensures that related functions are consolidated within the same chunked block. Despite the code being segmented, the logical association among related functions is preserved. This technique not only shortens the length of code fed into each prompt, thus addressing the token limitations but also maintains the code’s logical continuity, crucial for understanding the broader context. We attribute the success of the CCL-based chunking, in terms of handling code length issues and achieving commendable accuracy, to these methodological strengths.

To compare with the static analysis tools, we ran slither analyzer [39] on the smartBugs-curated dataset. The results show a detection rate equal to 11% which is equivalent to 23

⁶<https://zenodo.org/records/10530868>

detected vulnerable functions out of 207 vulnerable functions. This analysis highlights the effectiveness of our approach compared to the static analysis tools for vulnerability detection.

B. Adequacy of Our CAQ-Prompt in Practical Application

Our meticulously designed CAQ prompt is structured around four core components: judgment of vulnerabilities, explaining how the vulnerability can be exploited, business impact, and potential solutions. This structure ensures that the responses obtained are well-organized and present the comprehensive information one would expect in a professional audit report, thereby laying the groundwork for LLMs to generate thorough audit reports in the future.

Furthermore, by comparing the CAQ-based prompt with the traditional CWE-based prompt, we observed that the overall accuracy of the CAQ prompt is competitive, often requiring significantly less effort. For instance, in [19], the authors identified 38 types of vulnerabilities, implying that using a CAQ prompt could reduce the workload by up to 39 times compared to employing CWE-specific prompts. Given these advantages in terms of both efficiency and breadth of detection, we advocate for adopting CAQ-based prompts in LLM-based code vulnerability assessments.

C. Interpretation of the effectiveness of the CCL based chunking technique

The proposed approach aims to provide an efficient prompting technique that aims to reduce the prompt sizes, especially regarding complex codebase while providing GPT with enough data in order to analyze effectively the targeted functions and localize vulnerabilities. As discussed by authors in [62], As the input length increases, the model has to manage larger token numbers, which can lead to difficulties in maintaining coherence and relevance. This is particularly challenging in code-related tasks where precise understanding and execution of instructions are crucial. While models like GPT-4 have a high token limit, they still face challenges when approaching this limit. The need to chunk inputs or manage tokens effectively can result in loss of context or important details, impacting the quality of generated audits.

Besides, GPT-4 model relies on the self-attention mechanism to understand the relationships between words and their contexts. Self-attention allows each token to "attend" to all other tokens in the input sequence. However, this attention mechanism scales quadratically with the sequence length ($O(n^2)$ complexity). For long sequences, the amount of computational resources and memory required increases exponentially, making it harder for the model to effectively capture relationships [66].

As discussed in the section 4.3.2, findings 2 and 3 suggest that there is minimal disparity between the two prompting techniques and thus the most effective approach is the combination

of CCL-based chunking and the CAQ-based prompt. This can be explained by pattern recognition employed by GPT-4. GPT-4 is trained on vast datasets and is designed to recognize patterns and generalize information. Whether using CCL-based or CAQ-based prompting to find vulnerabilities in a smart contract, the LLM applies its understanding of common vulnerabilities and patterns it has learned for the training data.

The architecture of GPT models allows them to process and generate language based on learned patterns rather than specific rules. This means that for tasks like vulnerability detection, the model relies heavily on its training data’s statistical properties rather than a deterministic set of rules for pattern recognition [18].

D. Interpretation of the performance of GPT using CWE and CAQ prompt

The observed similarity in performance between CAQ prompts (generic prompts focusing on all vulnerabilities) and CWE prompts (specific prompts targeting particular vulnerabilities) can be attributed to the fundamental architecture and operation of large language models (LLMs). Transformer-based LLMs, such as OpenAI Codex, rely on identifying and applying patterns learned during pre-training to the input context. Generic prompts are likely activate a broad range of the model’s knowledge, encompassing both general and specific vulnerability patterns. This aligns with findings by [67], which demonstrated Codex’s ability to handle diverse programming tasks, revealing that generic queries often leverage the model’s extensive knowledge base effectively.

The retrieval mechanisms of LLMs further explain this overlap. Models tend to retrieve contextual patterns based on the input query while inherently including related peripheral information due to the interconnected structure of their training data. Even when tasked with identifying specific vulnerabilities, the model’s retrieval process draws upon a broader set of associated patterns. This phenomenon has been highlighted in studies like [68], which explore the dense interconnections within LLM training data, facilitating the retrieval of comprehensive knowledge irrespective of prompt specificity.

Additionally, prompt optimization research sheds light on this behavior. Studies such as [69] indicate that carefully crafted prompts can enhance performance in tasks requiring deep reasoning or nuanced outputs. However, tasks that depend heavily on pre-trained knowledge, such as vulnerability detection, tend to exhibit minimal sensitivity to prompt specificity. Generic prompts already activate the broad retrieval patterns necessary to address most specialized queries, resulting in limited differences in performance between general and specific prompts.

These findings suggest that LLMs exhibit a degree of prompt invariance, particularly in

domains where their pre-trained knowledge is expansive and well-integrated. For practical applications like smart contract auditing, this insight implies that focusing on robust generic prompts may suffice, reducing the need for extensive specialized prompt engineering. By leveraging the inherent strengths of LLMs, such an approach can streamline workflows while maintaining high task performance.

E. Implications on the proposed approach on industrial use

The primary objective of the proposed methodology is to facilitate expert security audits of smart contract projects. By leveraging CAQ-based prompts and CCL-based chunking, auditors receive a comprehensive assessment of potentially vulnerable functions, including explanations of proof-of-concept exploits and corresponding mitigation strategies. This structured information enhances the auditor’s understanding of individual function purposes within the broader contract and illuminates various exploitation pathways.

Experimental results demonstrate that CCL-based chunking effectively reduces the volume of code required for LLM prompts while maintaining sufficient context for accurate interpretation. This technique scales effectively to support professional auditors working with large code repositories.

3.4 Threat to Validity

- **Internal Threat**

A potential threat to the internal validity of our study arises from our automated script designed for extracting the Code Call List (CCL). Particularly when importing interfaces, complications may emerge. Due to various possible implementations for the same function signature, our algorithm might erroneously select an incorrect imported function. To alleviate this concern, we meticulously cross-examined all constructed CCLs, ensuring the correct implementation for each function was chosen. In real-world applications, smart contracts are often commented. The comments serve as a documentation of the written code. It also helps to enhance the code readability and clarity, which provides the developers and auditors with a deeper understanding of the aimed functionality of the code. For all these reasons, we did not remove the comments from the code. Our evaluation is performed on commented code. However, different smart contracts, including those obtained through reverse engineering, may not have comments, which may decrease the performance of the proposed approach. Another aspect to consider is the type of variables. Since the CCL is composed of a list of functions which can belong or not to the same contract, a special attention should be

given to the variables being used inside the functions. Functions belonging to different smart contracts can use the same name for state variables, and GPT is unable to distinguish the difference between the two variables since it does not have access to the state variable declaration area. In future works, we will be prompting that section along with the functions list.

- **External Threat**

Our evaluation leverages data from sources such as Smart Bugs and the SolidiFy-benchmark. These data contain labeled vulnerabilities and have been open-source since 2020. Given that these data might have been incorporated into GPT-4 code interpreter’s training, it poses a potential external threat. To mitigate this, we created a new dataset composed of unlabeled functions from the evaluation contracts. However, a lingering uncertainty remains: these functions might have been labeled by other datasets, though we have no confirmation of such overlaps.

Another threat is related to the usage of GPT-4 model. In our experiments, we did not execute our experiments using LLMs other than GPT-4. While this can be useful for the generalization of our results, we chose to focus solely on GPT-4 due to its state-of-the-art performance in code related tasks. Additionally, testing multiple LLMs would require extensive resources, which were beyond the scope of this study, but remain an important avenue for future work.

In our study, we define the divide the context into three components which are : code scoping, reporting scoping and assessment scoping. The context window is defined as the maximum number of tokens that an LLM can process in a single prompt. As code length increases, the complexity of interactions between different parts of the code also increases. LLMs’ performance for the task of auditing smart contracts decreases especially for long code chunks [62]. For this work, as the length of prompt is very important, we define the context of code as the code chunks and dependencies that are required for execution of the specific function we are auditing. While other works may have a different definition of the context, for this work we limit our definition of the context to the code chunks and dependencies a specific function requires to execute correctly. The non-deterministic nature of GPTs poses significant challenges and limitations, particularly in contexts where consistency and reliability are paramount. This inherent unpredictability arises from the models’ training on diverse and extensive datasets, which leads to a stochastic approach in generating responses. While this can yield creative and varied outputs, it also results in a lack of consistency, making the model’s reliability questionable in critical applications. Slither is used for code flatten-

ing and function call graph construction in the suggested approach. Slither supports one file flattening. We presume the created file and dependency management process are error-free. Complex dependencies might cause issues while flattening, resulting in unreachable code. All contracts that Slither fails to flatten are discarded.

3.5 Chapter Summary

In this chapter we present a novel approach to enhance the effectiveness of smart contract auditing using Large Language Models (LLMs), specifically GPT-4. We introduce a context-driven prompting technique that addresses key challenges in leveraging LLMs for smart contract co-auditing, such as handling long code contexts and providing relevant audit-specific information. The methodology employs three main techniques for context scoping and augmentation. First, code scoping is used to chunk long code into self-contained segments based on inter-dependencies, addressing the issue of lengthy code contexts. Second, assessment scoping enhances the context description based on the target assessment goal, effectively limiting the search space for vulnerabilities. Third, reporting scoping forces a specific format for the generated response, ensuring consistency and clarity in the audit reports. To guide the LLM in identifying vulnerabilities, we utilize the Common Weakness Enumeration (CWE) to reduce the search space and incorporate a set of defined Common Audit Questions (CAQ) extracted from previous audit reports. This approach helps scope the generated reports and focuses the LLM on relevant security concerns. Additionally, we developed a chunking approach based on code call graphs, called Code Call List (CCL)-based chunking, which provides sufficient code context without including the entire codebase within the prompt. The effectiveness of this approach was demonstrated through empirical evaluations on publicly available vulnerable contracts. The method achieved a remarkable 96% detection rate for vulnerable functions, significantly outperforming the native prompting approach, which only detected 53% of vulnerabilities. To further validate the reliability of our context-augmenting approach, the results were manually analyzed by expert auditors from Quantstamp, a leading smart contract auditing company. This expert analysis indicated that the proposed approach enhances GPT-4’s proficiency in detecting vulnerabilities in unlabeled datasets, showcasing its potential for real-world application in smart contract auditing.

CHAPTER 4 PRIORITIZING MUTANTS FOR EFFICIENT TEST SUITE GENERATION USING LLM

Software testing is a critical component of the software development lifecycle, aimed at ensuring the quality and reliability of software systems. As systems grow in complexity, the need for effective and efficient testing strategies becomes increasingly paramount. One of the primary challenges in developing efficient test suites is achieving comprehensive coverage while minimizing redundancy. As software systems become more complex, the number of possible execution paths and scenarios increases exponentially. This makes it difficult to create test cases that adequately cover all aspects of the system without becoming unwieldy or time-consuming to execute [70].

To address these challenges, researchers and practitioners have turned to advanced techniques such as search-based software testing (SBST) and heuristic approaches. SBST applies optimization algorithms, such as genetic algorithms, to automatically generate test cases that maximize coverage and fault detection while minimizing the overall size of the test suite. These techniques can explore vast search spaces more efficiently than manual approaches, potentially uncovering edge cases and vulnerabilities that might otherwise be missed. While such approaches have proven their efficiency in generating test cases that uncover edge cases, they require significant computational resources, especially for large and complex systems.

In recent years, mutation testing (MT) has emerged as a powerful technique for assessing and improving the quality of test suites while also guiding the generation of high-quality test cases for the program under test (PUT). Mutation testing is a software testing technique that involves modifying a program's source code to create faulty versions, known as mutants, to evaluate the effectiveness of a test suite in detecting errors. While the primary goal of mutation testing is to assess the quality of a test suite by determining whether it can identify and kill mutants, thereby ensuring the reliability and robustness of the software and unveiling edge cases, different approaches have been exploring the usage of surviving mutants to guide the generation of new test cases [71, 72]. Generating an adequate test suite using mutation testing can be computationally expensive and time-consuming due to the vast number of possible mutants. Furthermore, using mutants as test goals can lead to an exponential increase in the size of the test suite since each mutant will be killed by a test, and thus running the test suite can become impractical for complex codebases. Liu et al. introduced AID [73], a method combining LLMs with differential testing to generate fault-revealing test cases for programs that have already passed traditional test suites. This

approach demonstrates the potential of integrating advanced AI techniques with mutation testing to improve test case generation.

However, the generation of valid and effective test cases remains a challenge. The authors in [74] highlighted issues with correctness in LLM-generated test cases, while Yuan et al. proposed ChatTester [75], an approach that refines test cases iteratively to improve their quality. These studies underscore the need for sophisticated validation and refinement techniques when using AI-generated test cases.

In this chapter, we present a novel approach for incremental and adaptive test case generation using mutation testing. This approach enhances and evaluates the effectiveness of test cases generated by LLMs for Solidity smart contracts. Our approach aims to achieve effective test generation that optimizes the total number of generated tests while simultaneously enhancing the mutation score. Our proposed technique, PRiMG, employs two main components: a **mutation prioritization module**, responsible for selecting mutants as test goals for the generated test cases, and a **test case generation module**, which employs an LLM for generating and refining the test cases.

After an initial mutation testing campaign, we train a machine learning module leveraging the mutant subsumption graph [76] to predict the usefulness of a mutant as a test goal to generate a new test case. The test case generation process begins by selecting the prioritized mutant from a set of surviving mutants. The initial prompt includes the Program Under Test (PUT), a template test, the mutant code, and instructions for generating a test case that kills the mutant using a single-shot prompting strategy. Next, PRiMG employs a refining process that aims to verify the correctness of the generated test case regarding its syntax and pretest phase. The generated test case must pass the pretest phase, which includes successfully executing the program under test.

Subsequently, PRiMG applies mutation testing to evaluate the effectiveness of the test cases in killing the targeted mutant and other initially surviving mutants. We evaluated our approach on real-world Solidity projects obtained from Code4Arena, a CTF platform for blockchain security contests.

To summarize, this work makes the following contributions:

- We present a novel approach for incremental test suite generation using LLMs.
- We propose a mutation-based prioritization technique to guide test case generation, minimizing the size of the test suite while enhancing the mutation score.
- We assess the effectiveness of the generated tests in enhancing mutation scores on

real-world PUTs.

4.1 Approach

In this section, we discuss the different steps of our approach. Figure 4.1 shows an overview of our proposed approach, and Algorithm 2 presents the sequence of its different steps. PRiMG is composed of two main modules: **testcase generation module** and **mutants prioritization module**. We discuss each of the two modules in the following subsections.

Algorithm 2 PRiMG

Input: *PUT*, *initialTestSuite*, *survivingMutants*

Output: *finalTestSuite*

```

1: prioritizedMutant  $\leftarrow$  PrioritizeMutant(survivingMutants)
2: initTest  $\leftarrow$  SelectTest(prioritizedMutant, initialTestSuite)
3: initPrompt  $\leftarrow$  CONCAT(initTest, prioritizedMutant, PUT)
4: rawIUT  $\leftarrow$  LLM(initPrompt)
5: FUT, success  $\leftarrow$  RefiningTest(rawIUT, PUT, prioritizedMutant)
6: if success then
7:   Add FUT to finalTestSuite
8: end if
9: return finalTestSuite

```

4.1.1 Test Case Generation Module

Approach overview: This section details the process of generating test cases as outlined in Algorithm 2. Following an initial mutation testing phase, we identify and select the surviving mutants. The primary objective of the proposed approach is to develop unit tests capable of eliminating the maximum number of these surviving mutants. The process begins by selecting a specific mutant as the testing target, with the selection mechanism described in Section 4.1.2. The prioritized mutant is then used to generate an initial prompt for the large language model (LLM). Subsequently, the process enters a refinement phase, designed to ensure the correctness of the generated test cases, as elaborated in subsequent sections. If a test case is deemed correct, it is incorporated into the final test suite. Conversely, if the refinement process fails to produce a correct test case, the task is classified as problematic or one for which PRiMG is unable to generate an appropriate test case.

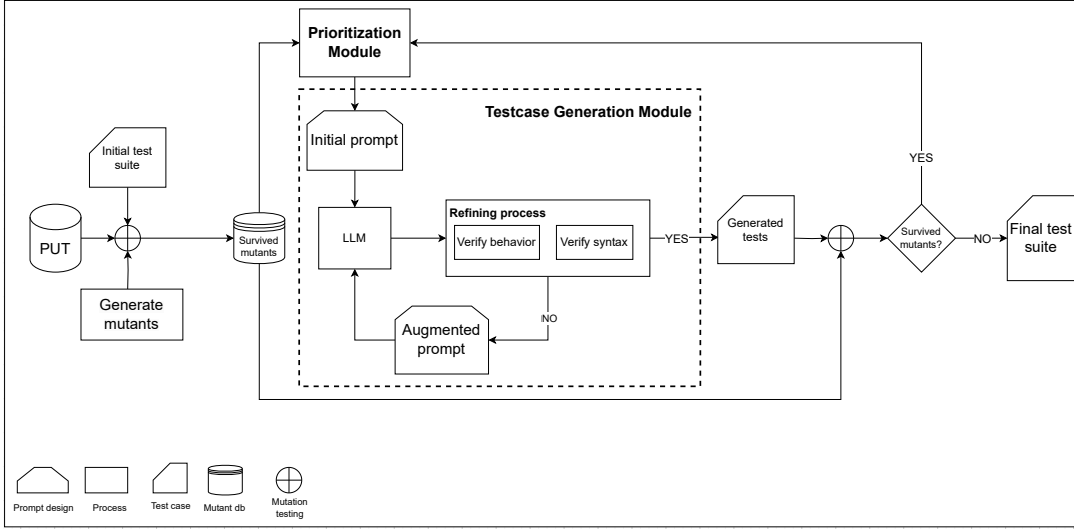


Figure 4.1 The proposed approach for generating a test case using LLMs.

Initial Prompt

LLMs are adept at performing tasks they have been trained on. However, fine-tuning an LLM to adapt to a new task is computationally expensive. Additionally, certain LLMs, such as Codex [67], excel in generating code but are closed-source, making fine-tuning for custom tasks infeasible.

Prompt-based learning [77, 78] offers an effective alternative for adapting LLMs to new tasks. A prompt is a combination of natural language and/or programming language context used as input to LLMs. Studies have demonstrated that incorporating natural language instructions as hints within prompts significantly enhances the ability of LLMs to perform novel tasks [79].

In our approach, the initial prompt consists of three key components:

- **The Program Under Test (PUT):** This represents the original program prior to mutation.
- **The Mutant Code:** This introduces modifications to the original program, representing potential faults or variations.
- **An Initial Test File:** This file is designed to validate the functionality and setup of the PUT within the testing framework. The test file is selected based on the following criteria:

Algorithm 3 RefiningTest

Input: *rawIUT*, *PUT*, *prioritizedMutant***Output:** *FUT*, *error*

```

1: correctTest  $\leftarrow$  false
2: FUT  $\leftarrow$  rawIUT
3: counter  $\leftarrow$  0
4: while not correctTest and counter < loopSize do
5:   syntaxVerif, errorSyntax  $\leftarrow$  Compile(FUT)
6:   behaviorVerif, errorBehavior  $\leftarrow$  RunTest(FUT, PUT)
7:   if syntaxVerif and behaviorVerif then
8:     correctTest  $\leftarrow$  true
9:   else
10:    augmentedPrompt  $\leftarrow$  CONCAT(FUT, PUT, errorSyntax, errorBehavior, prioritizedMutant)
11:    FUT  $\leftarrow$  LLM(augmentedPrompt)
12:    counter  $\leftarrow$  counter + 1
13:   end if
14: end while
15: return FUT, correctTest

```

1. The test file should test functions defined in the PUT. This ensures the large language model (LLM) receives sufficient information about the testing environment and the imported libraries required to execute the tests.
2. The test file should contain diverse test cases. This enhances the LLM's understanding of the structure and variety of test cases, aiding in the generation of comprehensive and accurate tests.

Refining Process

As illustrated in Algorithm 3, after generating the raw Initial Unit Test (IUT), we employ a refining process to enhance its quality and effectiveness. The refining process consists of two primary steps:

Syntax Verifier and Behavior Verifier In this step, we verify the correctness of the generated test case in two stages:

1. **Syntax Verification:** The test case is compiled, and logs are reviewed for syntax errors.

2. **Behavior Verification:** The test case is executed against the PUT. A valid test case must successfully pass when run on the PUT.

Prompt Augmentation If the previous step fails (e.g., the test case fails to compile or the PUT fails when tested), we proceed to prompt augmentation. In this step:

- A new prompt is constructed using the PUT, the mutant code, the FUT, and the error logs from the previous step.
- This enhanced prompt is used to generate a refined test case, referred to as the Final Unit Test (FUT), which resolves the issues encountered in the initial iteration.

By iteratively refining and augmenting the prompt, we ensure the generation of high-quality test cases capable of validating both the syntax and behavior of the program under test.

4.1.2 Prioritization module

Dynamic Mutant Subsumption Graph

In mutation testing, a test set T is considered *mutation-adequate* with respect to a mutant set M if for every non-equivalent mutant $m_i \in M$, there exists a test $t \in T$ such that t detects m_i [76]. However, mutation operators generate a significantly larger number of mutants than necessary, leading to redundancy. This redundancy is formally captured by the concept of *minimal mutation*. A dominator set $D \subseteq M$ is a minimal subset of M such that any test set adequate for D is also adequate for M .

A key aspect of mutation testing is the *subsumption relationship* between mutants. Given a finite set of mutants M and a finite test set T , a mutant m_i is said to *dynamically subsume* another mutant m_j if a test $t \in T$ detects m_i and every test that detects m_i also detects m_j . This relationship is *strict* if m_i subsumes m_j , but m_j does not subsume m_i . If two mutants m_i and m_j are detected by exactly the same tests in T , the subsumption is not strict [80].

The *Dynamic Mutant Subsumption Graph (DMSG)* captures these subsumption relationships. In a DMSG, each node represents a maximal set of redundant mutants, and edges represent the subsumption relationships between sets. Specifically, if mutant m_i strictly subsumes mutant m_j , there is a directed edge from the node containing m_i to the node containing m_j . Any test that detects a mutant in the DMSG is guaranteed to detect all mutants subsumed by that mutant, including those in the same node or lower in the graph. This structure provides an efficient way to identify and eliminate redundant mutants, thereby improving the mutation testing process and reducing the computational burden on developers.

Test Completeness Advancement Probability, TCAP for test case generation

Test Completeness Advancement Probability (TCAP), as defined in [80], is employed to measure mutant usefulness in the context of test case generation. This measure quantifies the likelihood of a mutant, when presented as a test goal to a developer, eliciting a test case that enhances test completeness by killing the mutant and a maximum number of other mutants. This captures a key idea: a mutant is useful to be selected as the test goal, if when developing a test case that kills that mutant, other mutants are going to be killed by the same developed test case. This is due to the fact that the subsumed mutant, m_j is killed by the same tests that kill the dominant mutant m_i . In a subsumption relationship, the test case that kills the dominant mutant is also killing the subsumed mutant.

As discussed in [80], we define the usefulness of a mutant in terms of its capacity to detect (or kill) other mutants when it is itself killed. However, for surviving mutants (mutants not killed by the initial test suite), it is not feasible to construct a Dominance Mutation Subsumption Graph (DMSG). Instead, TCAP exploits the structural information of the DMSGs of killed mutants. By analyzing these graphs, we can identify mutants that are likely to be useful as test goals. The intuition is that the test suite developed to kill these selected mutants will also effectively kill other mutants, thereby improving the overall efficiency of the test generation process and optimization the total number of needed tests. This approach builds on foundational ideas of mutation analysis and mutant subsumption as explored in prior research [76,81]. By leveraging the dominance relationships and subsumption hierarchies among mutants, we aim to streamline the test case generation process, improving both coverage and effectiveness through the usage of a supervised machine learning model.

The usefulness factor of a mutant is defined as measurement of its ability to reveal additional mutants when used as a target for test case generation. The usefulness factor of a mutant has three properties:

- Dominator mutants, have usefulness factor equal to 1: By definition, each test that kills a dominator mutants kills every other subsumed mutant.
- Subsumed mutants have a usefulness factor less than 1 and greater than 0

Machine learning module

- **context feature:**

Building upon the modeling approach and the feature set proposed by Samuel J. Kaufman et al. [80], we incorporated program context features extracted from a program's

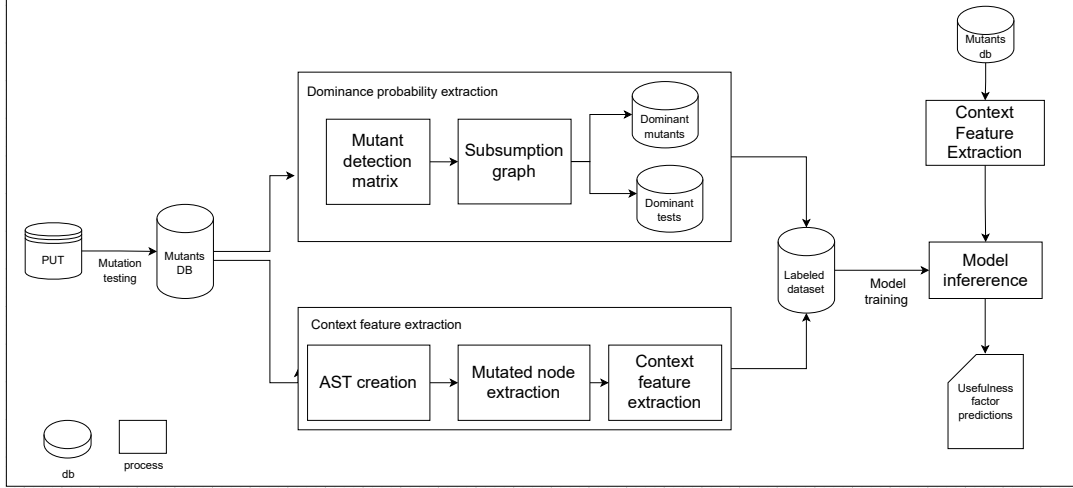


Figure 4.2 Overview of the dataset labeling process

Abstract Syntax Tree (AST). For a given mutated AST node, we examined its syntactic context, including parent and child nodes within the AST as well as nesting details, and its semantic context, such as the data types of expressions, operands, and method parameters. It is important to highlight that previous studies [82] utilized information derived from test execution data, such as code coverage, to make predictions in the context of mutation analysis. As done in [80], our approach aims to identify mutants that are beneficial for generating test cases that have not yet been created, aligning with the goals of mutation testing.

- **Feature extraction and dataset labeling:** Figure 4.2 shows an overview of our dataset labeling approach and model training. Since we can only construct the DMSG for the killed mutants, only the killed mutant will be used for training the model.

After an initial mutation testing campaign using the initial test suite of the PUT, we create our labeled dataset. The labeling of the dataset is composed of two modules: Dominance probability extraction and Context feature extraction. Given the mutation testing results, we create the mutation matrix which maps test cases to mutants, indicating which test cases can detect (kill) or not specific mutants. We use the mutation testing to construct the DMSG of mutants relative to the initial test suite as defined in the previous section. We extract the parent node which contains the dominant mutants. Those identified dominant mutants are labeled as useful having the useful factor equal to 1. The other mutants are labeled as having a useful factor equal to zero.

The second module is context feature creation module which aims to extract the context

features from the Abstract Syntax Tree (AST) described in the previous section. First we started by comparing the AST of the mutant with the one of the parent program. We identified the mutated node. Using the mutated node, we extracted the remaining context features of the mutated node and its parent node context features.

Using the usefulness factor and the context features, we created the labeled dataset that will be used for training and inference.

- **Machine learning models:** We evaluated a small number of model design choices and training settings against an intrinsic measure of model performance with the ultimate goal of choosing a single model for downstream evaluation. Each training setting was a kind of hold-one-out train-test split, using each held-out project (or smart contract in a given project) at a time as the evaluation set. We trained all models using Scikit-Learn [83] and evaluated every combination of the following choices:

Building on the work done by Samuel J. Kaufman et al. [80], we have chosen these configuration:

- **Model choice:** We compared ridge regression to a random forest regression [80, 84] (max depth of 3 and 10 trees). .
- **features set:** We used a comprehensive set of features (all features, enumerated in the previous section) [81]: only the mutation operator and detailed parent statement context).
- **Training Setting:** We evaluated the following train-test splits:
 - * *All Projects:* training on mutants from all projects, including those in non-held-out smart contracts in the same project, and testing on mutants in the held-out smart contracts;
 - * *Project-Only:* training on mutants in all but one held-out smart contract in a single project and testing on mutants in the held-out smart contract.

Building on findings in [80], we chose the following configuration for the experiments in the evaluation section:

Model=Linear, Features=All, Training Setting=Project-Only

4.2 Evaluation

In this section, we describe the evaluations we designed and conducted to investigate the following research questions.

1. **RQ1. How efficient is the pipeline for test case generation?**
2. **RQ2. How useful is the prioritization technique compared to random for efficient mutation testing?**

4.2.1 Experimental Setup

In this section, we present our experiment setup. Specifically, we describe the mutant generation tool, clarify the LLM of PRiMG and its setup, and indicate the benchmark data sets used in our experiments. We conducted the experiment on an EC2 instance of type g5.xlarge with 26 GiB of GPU memory.

Experiment Parameters

We call the refining process on the LLM up to 10 times and collect the output that meets two criteria: The generated test must compile correctly and then pass the PUT. If after 10 runs, the LLM is not able to generate an output that satisfies these two conditions, we consider the mutant as a problematic task or a task for which PRiMG is not able to generate a test case.

For the prioritization module, for each project, we trained a machine learning model to predict the survived mutant usefulness factor. As mentioned in 4.1.2, we used the configuration of: Linear model with few features. As we have already explained, the model is trained on mutants that have already been killed.

Mutant Generator

To apply mutation testing, we need to generate different mutant versions of a PUT by modifying different liens of code to introduce vulnerabilities. For this purpose we use SuMo [85] as mutant generation tool. SuMo is a mutation testing tool adapted for smart contracts of solidity. It uses different mutation operators to generate mutants including 25 solidity-specific operators and 19 general-purpose mutation operators.

For testing, we used Ganache¹ as local blockchain network to testing for deploying smart contracts and executing transactions for mutants and PUT. We use hardhat as testing framework².

¹Truffle Suite, *Ganache - One Click Blockchain*, <https://trufflesuite.com/ganache/>, accessed on Nov 26, 2024.

²Hardhat, <https://hardhat.org/>, accessed on Nov 26, 2024.

Data sources

Table 4.1 Data sources

Project Id	# of smart contracts	# of initial test files	Initial mutation score %
Allobase	31	12	35.89
Particle	19	7	46.86
Quadrata	10	9	77

To conduct our experiments, we chose real-world projects that we extracted from code4arena³. We selected three different projects. The details of the selected project is in the following Table 4.1. To ensue rigor, we chose projects meeting these specific criteria:

- Number of smart contracts: this criterion ensures the selection of complex projects by focusing on those with more than 10 smart contracts, avoiding overly straightforward examples.
- Initial test suite: in the selected project, the initial test suite must have at least one test case testing each smart contract.
- Mutation score: projects are chosen with an initial mutation score (MS) below 80%, ensuring that the existing test suite is insufficient and requires additional test cases to improve coverage.

4.2.2 Empirical Study Results

This section discusses our findings for each research question (RQ), focusing on the performance and efficiency of the proposed approaches.

Evaluation of the Performance of the Test Case Generation Module

RQ1. How efficient is the pipeline for test case generation?

In this research question, we evaluate the performance of the test case generation module with respect to two key aspects: (1) the effectiveness of refining module in generating correct test cases and (2) the ability of these generated tests to kill new mutants that were not initially addressed by the original test suite. To ensure a comprehensive analysis, RQ1 is divided into the following sub-questions:

³<https://code4rena.com>

- **RQ1.1:** How efficient is the pipeline for generating syntactically and behaviorally correct tests?

The objective here is to assess the accuracy of the test cases generated by the LLM. Correctness is evaluated on two dimensions: **syntactic correctness**: The test case compiles successfully, and **behavioral correctness**: The test case passes the original Program Under Test (PUT) without errors.

- **RQ1.2:** What is the impact of the size of the refining loop on the correctness of the generated tests?

This sub-question investigates how the number of iterations in the refining loop impacts the quality of the generated test cases. Specifically, it examines whether iterative improvements significantly enhance the syntactic and behavioral correctness of the test cases.

To answer RQ1.1, test cases were generated by prompting the LLM with and without the refining process using different loop sizes. A test case is considered correct if it successfully compiles and passes the original PUTs. The results were collected from three projects, as listed in Table 4.1.

To answer RQ1.1 and statistically evaluate the impact of the refining process, we formulated the following hypotheses [63]:

RQ1.1-H0: *The refining process does not significantly improve the number of correct tests compared to a single-shot prompt.*

RQ1.1-H1: *The refining process significantly improves the number of correct tests compared to a single-shot prompt.*

To evaluate the efficiency of the test generation process, we analyzed the performance of test case generation with respect to the number of correctly generated test cases across varying trial counts. Table 4.3 summarizes the results obtained when running the process with 1 trial, 5 trials, and 10 trials. To assess the statistical significance of the differences between these configurations, we performed pairwise Z-tests for proportions, a statistical method suitable for comparing success rates across independent trials.

The **Z-test** is a statistical analysis method commonly used to determine whether there is a significant difference between sample and population means or between two sample means. It is particularly appropriate when the population is normally distributed or when the sample size is sufficiently large (typically $n \geq 30$), as the Central Limit Theorem ensures that the sampling distribution of the mean approaches normality under these conditions. The z-

test assumes that the population standard deviation (σ) is known, which provides a reliable estimate of variability and enables the calculation of standardized test statistics. This makes the z-test ideal for large-sample scenarios where the normal approximation holds true and reduces the influence of variability or outliers. By leveraging these properties, the z-test provides robust and accurate results for hypothesis testing, confidence interval estimation, and other inferential statistical tasks, ensuring its widespread applicability in research and data analysis.

The comparisons were conducted across three projects: Allobase, Particle, and Quadrata. The results of the Z-tests for each project are as follows:

- **Allobase:** Significant differences were observed between 1 trial and 5 trials ($Z = -19.08, p < 0.05$) as well as 1 trial and 10 trials ($Z = -19.20, p < 0.05$). No significant difference was found between 5 trials and 10 trials ($Z = -0.22, p > 0.05$).
- **Particle:** Significant differences were observed between 1 trial and 5 trials ($Z = -12.13, p < 0.05$) and 1 trial and 10 trials ($Z = -12.13, p < 0.05$). No significant difference was found between 5 trials and 10 trials ($Z = 0.0, p = 1.0$).
- **Quadrata:** Significant differences were observed between 1 trial and 5 trials ($Z = -10.24, p < 0.05$) and 1 trial and 10 trials ($Z = -10.54, p < 0.05$). No significant difference was found between 5 trials and 10 trials ($Z = -0.35, p = 0.73$).

Project	Single-Shot Correct	Refining Process (Loop of 5)	Refining Process (Loop of 10)
Allobase	23/500	201/500	203/500
Particle	10/500	151/500	151/500
Quadrata	21/500	140/500	145/500

Table 4.2 Number of correct test cases generated in different projects using different loop sizes

Specifically, for the projects Allobase, Particle, and Quadrata, the Z-test results provide strong evidence that increasing the number of trials significantly enhances the success rates of generating correct tests. These findings suggest that using more trials contributes to better test generation performance, as evidenced by the statistically significant improvements observed when increasing from 1 trial to 5 trials and from 1 trial to 10 trials. However, no significant differences were found between 5 and 10 trials across all projects, indicating diminishing returns beyond 5 trials. The observed differences in success rates are statistically significant under the experimental conditions and datasets analyzed. Thus, increasing the

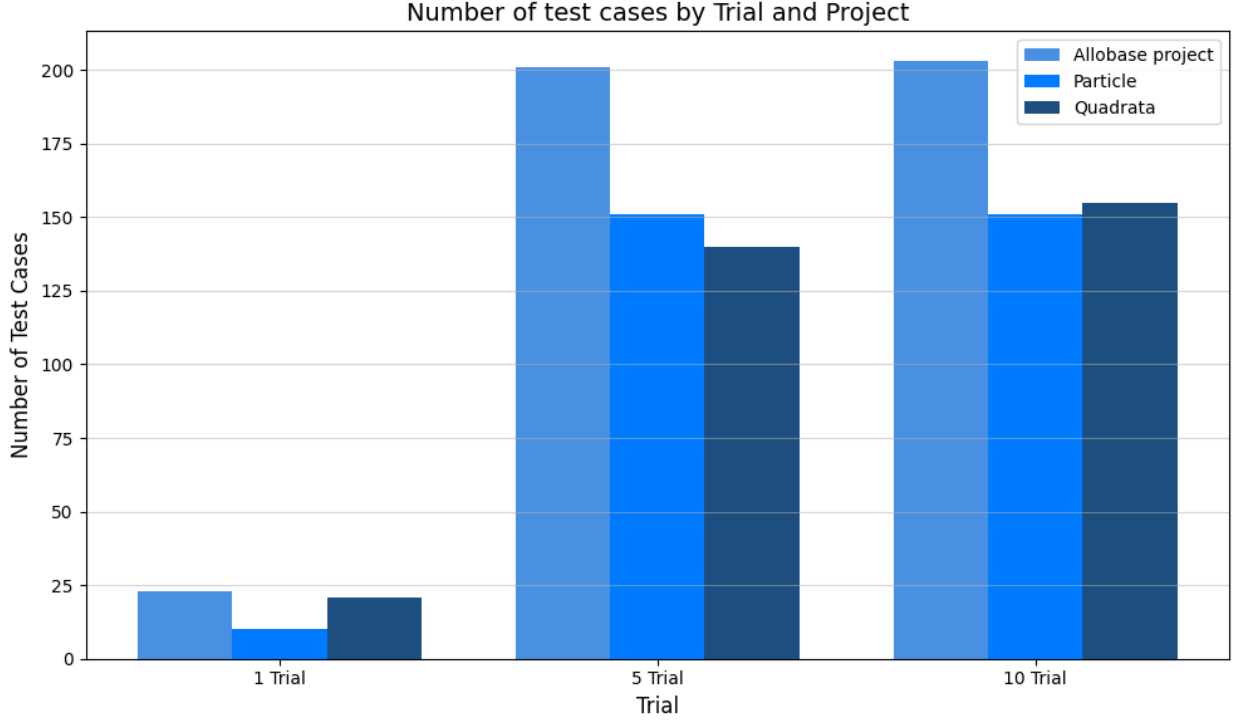


Figure 4.3 Number of test cases by trial and projects

number of trials, particularly from 1 to 5, outperforms single-shot attempts in generating correct tests, highlighting the value of incorporating multiple trials in improving LLM-generated test cases.

Additionally, as shown in Table 4.2, the impact of the refining process on the performance of test case generation is evident. In the *Allobase* project, the single-shot approach generates only 4.6% of correct tests, whereas using the refining process with a loop size of 5 iterations results in a substantial improvement, with 40.2% of tests being correct. Similarly, for the *Particle* project, the single-shot approach yields only 2% correct tests, while applying the refining process with 5 iterations increases the correct tests to 30.2%. In the case of *Quadrata*, the single-shot approach results in 4.2% correct tests, and refining with 5 iterations improves this to 28%. These results indicate a significant performance boost when employing the refining module, demonstrating that iterative refinement is a crucial factor in improving the accuracy and effectiveness of the test case generation module. Therefore, we conclude that utilizing the refining module greatly enhances the performance of the test case generation process, making it a valuable strategy for generating higher-quality test cases in all evaluated projects.

Finding 1: The refining module can significantly improve the correctness of the generated tests.

To answer **RQ1.2**, we conducted an additional analysis focusing on the effect of loop size on the correctness of the generated tests. In this research question, our primary objective is to evaluate the role of the number of iterations (loop size) within the refining module on the overall performance of the test case generation process. The hypothesis guiding this investigation was based on the premise that increasing the number of loops could lead to more refined and accurate test cases. The results of this analysis, including detailed performance comparisons for each project, are presented in Table 4.2.

We formulated the null hypothesis (H0) and alternative hypothesis (H1) to test the impact of the number of trials as follows:

RQ1.2-H0: *Both 5 and 10 loops significantly improve the number of correct test cases compared to single-shot for all projects.*

RQ1.2-H1: *Increasing the number of trials in the test generation process positively impacts the success rate of generated test cases.*

To analyze the performance of the test generation module across different trial configurations (1 trial, 5 trials, and 10 trials), we employed pairwise Z-tests for proportions. These tests compare the success rates of correctly generated tests for different numbers of trials. The Z-test was applied to all three projects, Allobase, Particle, and Quadrata, and the results are summarized below:

- **Allobase:** No significant difference was found between 5 and 10 trials ($Z = -0.22, p > 0.05$).
- **Particle:** No significant difference was found between 5 and 10 trials ($Z = 0.00, p > 0.05$).
- **Quadrata:** No significant difference was found between 5 and 10 trials ($Z = -0.35, p > 0.05$).

These findings suggest that increasing the number of trials from 1 to 5 significantly improves the success rates of test case generation across all projects. However, further increases beyond 5 trials do not result in statistically significant improvements. The Z-test results emphasize the value of employing multiple trials, particularly up to 5, to enhance the accuracy and reliability of LLM-generated test cases.

Finding 2: No significant improvement is observed when increasing from 5 to 10 loops, suggesting diminishing returns beyond 5 iterations.

Overall, the results suggest that the refining loop technique using a loop size of 5 is the best configuration for optimal test case generation performance.

Evaluation of the performance of prioritization module

RQ2. How useful is the prioritization technique compared to random for efficient mutation testing?

To evaluate the prioritization module, we perform analysis on the tests generated for the projects as mentioned in 4.3. For each project among the three projects, we generated 500 tests using the test case generation module. We train a linear model for each project as described in section 4.1.2. We use the configuration *Model=Linear, Features=all, Training Setting=Project-Only*.

We distinguish between 2 types of tests: (1) Prioritized test is a test case that have been generating targeting a prioritized mutant and (2) random test is test that have been generated targeting a random mutant.

In Table 4.3 we illustrate the results of the mutation testing we executed using three random set of tests composed of 50 tests and one prioritized set of tests. To answer RQ2, we conducted a hypothesis test on the results project Allobase, Particle and Quadrata as presented in Table 4.3. We formulated the null hypothesis (H0) and the alternative hypothesis (H1) as follows:

RQ2-H0: *There is no statistically significant difference in the performance of prioritized generated tests compared to randomly generated tests*

RQ2-H1: *There is a statistically significant difference in the performance of prioritized generated tests compared to randomly generated tests*

Similar to RQ1.1 and RQ1.2, we compare the efficacy of these strategies (i.e., prioritized tests vs. randomly selected tests) by using Z-test.

- **Allobase:** The Z-test for this comparison yielded $Z = -15.98, p < 0.05$, indicating a statistically significant difference between the two methods, with Method 2 outperforming Method 1.
- **Particle:** The Z-test for this comparison yielded $Z = -12.94, p < 0.05$, also showing a

statistically significant difference between the two methods, with Method 2 performing better than Method 1.

- **Quadrata:** The Z-test for this comparison yielded $Z = -14.12, p < 0.05$, demonstrating a statistically significant difference between the two methods, with Method B outperforming Method A.

These results suggest that, across all three projects, Allobase, Particle, and Quadrata, the test suite composed of prioritized tests significantly outperforms the test suite composed of random test sets generated using random selection of mutants in killing mutants. The observed differences in success rates are statistically significant under the experimental conditions analyzed, highlighting the effectiveness of the prioritization technique in killing mutants. Thus, the prioritization module consistently enhances test performance, demonstrating its superiority over random selection in all the evaluated projects. These results highlight the potential of the prioritization strategy as a valuable tool in mutation testing, contributing to more efficient and targeted test generation in software testing practices.

Finding 3: There is a statistically significant difference in the performance of prioritized generated tests compared to randomly generated tests in killing mutants.

Project	Randomly Selected Test - 1	Randomly Selected Test - 2	Randomly Selected Test - 3	Prioritized Test	Ground Truth
Allobase	200	158	8	300	465
Particle	196	54	180	250	350
Quadrata	55	205	200	256	403

Table 4.3 Number of killed mutants using Randomly Selected Tests, Prioritized Test, and Ground Truth

4.3 Discussions

4.3.1 Interpretation of the importance of the loop size on the effectiveness of test case generation

Although the refining module has shown impressive results in generating and refining generated tests, experiments in the evaluation section show that increasing the size of the loop does not considerably improve the number of correctly generated tests since even having a larger trial number, the LLM still fails to generate correct tests no matter the number of repetitions. In fact, LLMs operate based on statistical patterns in their training data rather than true understanding of programming concepts or software semantics [86]. This

can lead to what researchers call "hallucination" - generating plausible-looking but incorrect code that doesn't actually fix the underlying bug [87]. As Dou et al. observed in their extensive study, functional bugs constitute the highest proportion of errors in LLM-generated code fixes, with misunderstanding and logic errors being particularly prevalent [86]. Even after multiple attempts, an LLM may continue to make similar mistakes if it fundamentally misunderstands the problem or lacks the specific knowledge needed to generate a correct fix. Besides, the token-by-token generation process of LLMs can lead to inconsistencies in longer code sequences, even if individual parts seem correct. This is particularly problematic for bug fixes that require changes across multiple lines or functions. The authors in [86] observed that LLMs tend to perform better on localized, single-line fixes but struggle with bugs that require broader context or multiple coordinated changes. Furthermore, the limited context window of most LLMs can prevent them from fully considering all relevant parts of a large codebase, leading to fixes that may solve one issue but introduce new bugs elsewhere. These structural limitations of LLMs mean that even with repeated attempts, they may struggle to generate consistently correct and comprehensive bug fixes for more complex software issues.

4.3.2 Adequacy of our approach for practical applications

The PRiMG framework presents a practical and efficient solution for test suite generation by integrating automated test case generation with an advanced prioritization module. Leveraging Large Language Models (LLMs), the framework automates the creation of diverse, high-quality test cases, minimizing the manual effort required from developers while ensuring robust coverage of edge cases and complex scenarios. Complementing this, the prioritization module employs machine learning to rank and focus on the most impactful mutants, optimizing the testing process by reducing redundant efforts and maintaining high mutation coverage with a smaller, more efficient test suite. This dual approach significantly enhances the efficiency of mutation testing, streamlining the traditionally resource-intensive process and enabling developers to concentrate on critical tests. Particularly in domains like smart contract development, where rapid and reliable testing is crucial, PRiMG demonstrates its value by reducing computational overhead and improving the precision and practicality of the testing workflow. Its combination of automation, prioritization, and scalability establishes PRiMG as a powerful tool for advancing software quality in both research and industrial contexts.

4.4 Threat to Validity

In this study, we employed two different prompt-based learning techniques: one-shot and prompting augmenting technique. However, we did not explore the potential impact of altering the natural language instructions or demonstrative examples (for the prompt augmenting technique) within our prompts. Modifying these instructions or utilizing different demonstrative examples more closely aligned with the PUT’s functionality could potentially enhance the results.

A potential threat to the validity of our study arises from the choice of mutation operators defined within the SuMo framework [85]. Mutation operators play a critical role in generating test cases, and the limited set of operators available in SuMo may not adequately represent all possible vulnerabilities in smart contracts. This can lead to a biased set of mutants, limiting the effectiveness of the generated tests in detecting a broad range of issues. If operators do not cover diverse attack vectors or edge cases, the results of mutation testing may not fully reflect the real-world security risks present in smart contracts, potentially compromising the generalizability of our findings.

Another validity threat arises from the exclusive use of Hardhat as the testing framework. While Hardhat is a widely adopted and robust tool for smart contract development and testing, its exclusive use may restrict the applicability of our results to other testing environments. Variations in how different frameworks execute and interact with smart contracts could lead to discrepancies in test outcomes. As a result, the conclusions drawn from Hardhat-based testing might not extend to projects using alternative frameworks, thus reducing the external validity of our study.

Additionally, this study did not evaluate the number of newly killed mutants or the impact of the new tests on the overall mutation score. Although this metric is important, our primary focus was on generating correct and valid test cases.

A further threat to validity arises from the practical challenges and resource limitations encountered during testing. While there is no inherent limitation in the availability of projects, the complexities involved in setting up the testing environment and the constrained resources for conducting extensive experiments restricted the scope of our analysis. Overcoming these challenges in future work could enable a broader evaluation across more diverse projects and environments.

4.5 Chapter Summary

In this chapter, we present PRiMG, a novel approach for incremental test suite generation using large language models (LLMs) to enhance the quality of testing in Solidity-based smart contracts. PRiMG integrates a mutation-based prioritization technique to guide the generation of test cases, aiming to minimize the size of the test suite while maximizing its effectiveness in improving mutation scores. The approach employs mutation testing to evaluate the capability of the generated test cases to kill both targeted mutants and other initially surviving mutants, providing a comprehensive assessment of their utility. To validate the effectiveness of PRiMG, we conducted experiments on real-world Solidity projects sourced from Code4Arena, a Capture the Flag (CTF) platform for blockchain security contests. Our evaluation demonstrates that PRiMG significantly improves the mutation scores for real-world programs under test (PUTs), showcasing its potential to enhance the robustness and efficiency of smart contract testing.

CHAPTER 5 CONCLUSION

5.1 Summary of Works

This research contributes to improve the security and efficiency of smart contract auditing, addressing the growing complexity of blockchain applications. By focusing on the challenges of vulnerability detection, we developed a context-driven prompting technique for Large Language Models (LLMs) which optimizes their ability to identify vulnerabilities in smart contracts. Through the use of Code Scoping, Assessment Scoping, and Reporting Scoping, this approach enhances the LLMs' accuracy and relevance, thus supporting human auditors in focusing on the most critical areas of the code.

In addition to enhancing vulnerability detection, we introduced PRiMG, a prioritization technique that improves test case generation for Solidity smart contracts using mutation testing. By incorporating both a mutation prioritization module and a test case generation module, PRiMG helps ensure that test cases are effectively prioritized, allowing auditors to address the most pressing vulnerabilities first. The validation of PRiMG through real-world projects demonstrated its potential to significantly optimize the auditing process and enhance the reliability of smart contracts.

Together, these contributions offer a robust framework for smart contract auditing that combines human expertise with automated tools. The techniques we developed are timely and crucial, given the growing adoption of blockchain technologies across industries. As the security of smart contracts remains a critical factor for the successful implementation of decentralized systems, our research offers innovative solutions to ensure that smart contracts are secure, trustworthy, and suitable for widespread application.

While our research primarily focused on smart contracts and the Solidity language, the methodologies and techniques developed in this work have broader applications across various programming languages and software domains. The context-driven prompting technique for LLMs can be adapted to analyze vulnerabilities in other programming languages, as the underlying principles of code structure, security patterns, and vulnerability types often share commonalities across different languages. Similarly, the PRiMG prioritization technique, while tailored for Solidity smart contracts, presents a framework that can be extended to other languages by adjusting the mutation operators and test case generation strategies to suit the specific language constructs and common vulnerability patterns.

The generalizability of these approaches is particularly relevant in the broader context of

software engineering and security. As software systems grow in complexity across various domains, from web applications to embedded systems, the need for efficient and accurate vulnerability detection and testing prioritization becomes increasingly critical. By adapting the context-driven prompting and prioritization techniques to different programming paradigms and environments, researchers and practitioners can potentially enhance security auditing processes across a wide range of software projects, ultimately contributing to the overall improvement of software quality and security in diverse technological landscapes.

5.2 Limitations and Future Works

In this section, we discuss the limitations of our studies and potential directions for future research.

- In the area of LLM-base smart contracts auditing, future research can focus on several key directions to enhance the effectiveness and reliability of these systems. One promising avenue is the development of domain-specific pre-training and multi-stage fine-tuning techniques tailored to smart contract languages like Solidity, enabling LLMs to better understand the nuances of blockchain vulnerabilities. Additionally, integrating LLMs with traditional static and dynamic analysis tools could create hybrid frameworks that leverage the strengths of both approaches, improving vulnerability detection and coverage. Another critical area is the advancement of automated patch generation, where LLMs not only identify vulnerabilities but also propose and validate fixes, streamlining the remediation process. To address scalability challenges, optimized model architectures and parallel processing techniques could be explored to handle large-scale audits more efficiently. Furthermore, enhancing explainability through detailed reasoning trails and visual representations of vulnerabilities would improve trust and usability for human auditors. Lastly, fostering human-AI collaboration by designing interactive auditing interfaces and adaptive learning systems could ensure that LLMs continuously improve while supporting auditors in real-time, ultimately advancing the security and reliability of blockchain ecosystems.
- Future works could focus on extending the PRiMG framework to support additional programming languages and incorporate newer Large Language Models (LLMs) with improved capabilities for handling complex logic and broader context. Exploring alternative refinement strategies, such as leveraging reinforcement learning to dynamically adapt the test case generation process, may enhance the efficiency and correctness of the generated tests. Furthermore, addressing limitations in mutation operator diversity

and expanding the dataset to include a wider range of real-world projects could improve the generalizability of the results. Investigating the integration of PRiMG with other blockchain testing frameworks or CI/CD pipelines would also offer practical benefits, facilitating its adoption in professional environments. Finally, a deeper analysis of the interplay between mutant prioritization and test effectiveness in diverse scenarios could provide valuable insights to further optimize the test generation process.

REFERENCES

- [1] T. De Graaf, “From old to new: From internet to smart contracts and from people to smart contracts,” *Computer law & security review*, vol. 35, no. 5, p. 105322, 2019.
- [2] Z. Zheng *et al.*, “An overview on smart contracts: Challenges, advances and platforms,” *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020.
- [3] M. Kolvart, M. Poola, and A. Rull, “Smart contracts,” in *The Future of Law and etech-nologies*. Springer, 2016, pp. 133–147.
- [4] P. Zhang, F. Xiao, and X. Luo, “A framework and dataset for bugs in ethereum smart contracts,” in *2020 IEEE International Conference on Software Maintenance and Evo-lution (ICSME)*. IEEE, 2020, pp. 139–150.
- [5] C. Carpentier-Desjardins *et al.*, “Mapping the defi crime landscape: An evidence-based picture,” *arXiv preprint arXiv:2310.04356*, 2023.
- [6] Z. Zhioua, S. Short, and Y. Roudier, “Static code analysis for software security verifica-tion: Problems and approaches,” in *2014 IEEE 38th International Computer Software and Applications Conference Workshops*. IEEE, 2014, pp. 102–109.
- [7] A. M. Rozario and M. A. Vasarhelyi, “Auditing with smart contracts.” *International Journal of Digital Accounting Research*, vol. 18, 2018.
- [8] D. He *et al.*, “Smart contract vulnerability analysis and security audit,” *IEEE Network*, vol. 34, no. 5, pp. 276–282, 2020.
- [9] I. Garfatta *et al.*, “A survey on formal verification for solidity smart contracts,” in *Proceedings of the 2021 Australasian Computer Science Week Multiconference*, 2021, pp. 1–10.
- [10] W. Zou *et al.*, “Smart contract development: Challenges and opportunities,” *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2084–2106, 2019.
- [11] F. Mi *et al.*, “Vscl: automating vulnerability detection in smart contracts with deep learning,” in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2021, pp. 1–9.
- [12] S. Qian *et al.*, “Multi-label vulnerability detection of smart contracts based on bi-lstm and attention mechanism,” *Electronics*, vol. 11, no. 19, p. 3260, 2022.

- [13] Y. Sun *et al.*, “Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
- [14] B. Boi, C. Esposito, and S. Lee, “Vulnhunt-gpt: a smart contract vulnerabilities detector based on openai chatgpt,” in *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, 2024, pp. 1517–1524.
- [15] S. Zeng *et al.*, “Solgpt: A gpt-based static vulnerability detection model for enhancing smart contract security,” in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2023, pp. 42–62.
- [16] W. Ma *et al.*, “Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications,” *arXiv preprint arXiv:2403.16073*, 2024.
- [17] E. A. Napoli and V. Gatteschi, “Evaluating chatgpt for smart contracts vulnerability correction,” in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2023, pp. 1828–1833.
- [18] C. Chen *et al.*, “When chatgpt meets smart contract vulnerability detection: How far are we?” *arXiv preprint arXiv:2309.05520*, 2023.
- [19] I. David *et al.*, “Do you still need a manual smart contract audit?” *arXiv preprint arXiv:2306.12338*, 2023.
- [20] Z. Wang *et al.*, “Efficiently detecting reentrancy vulnerabilities in complex smart contracts,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 161–181, 2024.
- [21] Z. Wei *et al.*, “Llm-smartaudit: Advanced smart contract vulnerability detection,” *arXiv preprint arXiv:2410.09381*, 2024.
- [22] V. Mothukuri, R. M. Parizi, and J. L. Massa, “Llmsmartsec: Smart contract security auditing with llm and annotated control flow graph,” in *2024 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2024, pp. 434–441.
- [23] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.
- [24] G. Petrovic *et al.*, “An industrial application of mutation testing: Lessons, challenges, and research directions,” in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 47–53.

- [25] K. Li *et al.*, “Static application security testing (sast) tools for smart contracts: How far are we?” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, p. 1447–1470, Jul. 2024. [Online]. Available: <http://dx.doi.org/10.1145/3660772>
- [26] M. Fooladgar and F. Faghieh, “Effective targeted testing of smart contracts,” 2024. [Online]. Available: <https://arxiv.org/abs/2407.04250>
- [27] Y. Liu *et al.*, “Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation,” *arXiv preprint arXiv:2405.02580*, 2024.
- [28] R. Karanjai *et al.*, “Generating move smart contracts based on concepts,” *arXiv preprint arXiv:2412.12513*, 2024.
- [29] S. Chatterjee and B. Ramamurthy, “Efficacy of various large language models in generating smart contracts,” *arXiv preprint arXiv:2407.11019*, 2024.
- [30] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
- [31] S. Nakamoto, “Bitcoin whitepaper,” *URL: https://bitcoin.org/bitcoin.pdf-(17.07.2019)*, vol. 9, p. 15, 2008.
- [32] G. Wu *et al.*, “A comprehensive survey of smart contract security: State of the art and research directions,” *Journal of Network and Computer Applications*, p. 103882, 2024.
- [33] M. Liu *et al.*, “I experienced more than 10 {DeFi} scams: On {DeFi} users’ perception of security breaches and countermeasures,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 6039–6055.
- [34] S. Ren *et al.*, “Lookahead: Preventing defi attacks via unveiling adversarial contracts,” *arXiv preprint arXiv:2401.07261*, 2024.
- [35] A. Fan *et al.*, “Large language models for software engineering: Survey and open problems,” *arXiv preprint arXiv:2310.03533*, 2023.
- [36] X. Hou *et al.*, “Large language models for software engineering: A systematic literature review,” *arXiv preprint arXiv:2308.10620*, 2023.
- [37] N. Sharma and S. Sharma, “A survey of mythril, a smart contract security analysis tool for evm bytecode,” *Indian J Natural Sci*, vol. 13, p. 75, 2022.

- [38] M. Mossberg *et al.*, “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.
- [39] J. Feist, G. Grieco, and A. Groce, “Slither: a static analysis framework for smart contracts,” in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.
- [40] L. Luu *et al.*, “Making smart contracts smarter,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.
- [41] T. Durieux *et al.*, “Empirical review of automated analysis tools on 47,587 ethereum smart contracts,” in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.
- [42] A. Ghaleb and K. Pattabiraman, “How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 415–427.
- [43] X. Chen, “Crossinspector: A static analysis approach for cross-contract vulnerability detection,” *arXiv preprint arXiv:2408.15292*, 2024.
- [44] D. Li *et al.*, “Smart contract vulnerability detection based on static analysis and multi-objective search,” *arXiv preprint arXiv:2410.00282*, 2024.
- [45] D. Ressi *et al.*, “Vulnerability detection in ethereum smart contracts via machine learning: A qualitative analysis,” *arXiv preprint arXiv:2407.18639*, 2024.
- [46] T. Zhou *et al.*, “Smartgift: Learning to generate practical inputs for testing smart contracts,” in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 23–34.
- [47] H. G. Gurbuz *et al.*, “Test suite assessment of safety-critical systems using safety tactics and fault-based mutation testing,” *Cluster Computing*, pp. 1–25, 2024.
- [48] S. Ji *et al.*, “Test-case generation for data flow testing of smart contracts based on improved genetic algorithm,” *IEEE Transactions on Reliability*, vol. 72, no. 1, pp. 358–371, 2022.
- [49] S. Driessen *et al.*, “Automated test-case generation for solidity smart contracts: The agsolt approach and its evaluation,” *arXiv preprint arXiv:2102.08864*, 2021.

- [50] M. Olsthoorn *et al.*, “Syntest-solidity: Automated test case generation and fuzzing for smart contracts,” in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 202–206.
- [51] A. Panichella, F. M. Kifetew, and P. Tonella, “Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets,” *IEEE Transactions on Software Engineering*, vol. 44, pp. 122–158, 2018. [Online]. Available: <https://api.semanticscholar.org/CorpusID:3738201>
- [52] W. Zou *et al.*, “Smart contract development: Challenges and opportunities,” *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2084–2106, 2021.
- [53] W. C. Ouédraogo *et al.*, “Large-scale, independent and comprehensive study of the power of llms for test case generation,” *arXiv preprint arXiv:2407.00225*, 2024.
- [54] A. M. Dakhel *et al.*, “Effective test generation using pre-trained large language models and mutation testing,” *Information and Software Technology*, vol. 171, p. 107468, 2024.
- [55] Z. Zheng *et al.*, “Towards an understanding of large language models in software engineering tasks,” *arXiv preprint arXiv:2308.11396*, 2023.
- [56] S. Sarsa *et al.*, “Automatic generation of programming exercises and code explanations using large language models,” in *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*, 2022, pp. 27–43.
- [57] H. Joshi *et al.*, “Repair is nearly generation: Multilingual program repair with llms,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, 2023, pp. 5131–5140.
- [58] J. Li *et al.*, “Loogle: Can long-context language models understand long contexts?” *arXiv preprint arXiv:2311.04939*, 2023.
- [59] M. X. Liu *et al.*, ““what it wants me to say”: Bridging the abstraction gap between end-user programmers and code-generating large language models,” in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–31.
- [60] A. Krishnamurthy *et al.*, “Can large language models explore in-context?” *arXiv preprint arXiv:2403.15371*, 2024.
- [61] Y. Zhu *et al.*, “Can large language models understand context?” *arXiv preprint arXiv:2402.00858*, 2024.

- [62] M. Levy, A. Jacoby, and Y. Goldberg, “Same task, more tokens: the impact of input length on the reasoning performance of large language models,” *arXiv preprint arXiv:2402.14848*, 2024.
- [63] C. Wohlin *et al.*, *Experimentation in software engineering*. Springer Science & Business Media, 2012.
- [64] M. Q. Pembury Smith and G. D. Ruxton, “Effective use of the mcnemar test,” *Behavioral Ecology and Sociobiology*, vol. 74, pp. 1–9, 2020.
- [65] M. L. McHugh, “Interrater reliability: the kappa statistic,” *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.
- [66] X. Wang *et al.*, “Beyond the limits: A survey of techniques to extend the context length in large language models,” *arXiv preprint arXiv:2402.02244*, 2024.
- [67] M. Chen *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [68] J. Yuan *et al.*, “Towards a holistic evaluation of llms on factual knowledge recall,” *arXiv preprint arXiv:2404.16164*, 2024.
- [69] J. Cheng *et al.*, “Black-box prompt optimization: Aligning large language models without model training,” *arXiv preprint arXiv:2311.04155*, 2023.
- [70] P. McMinn, “Search-based software testing: Past, present and future,” in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 153–163.
- [71] F. Tip, J. Bell, and M. Schäfer, “Llmorpheus: Mutation testing using large language models,” *arXiv preprint arXiv:2404.09952*, 2024.
- [72] H. Du, V. K. Palepu, and J. A. Jones, “Leveraging propagated infection to crossfire mutants,” *arXiv preprint arXiv:2411.09846*, 2024.
- [73] K. Liu *et al.*, “Llm-powered test case generation for detecting tricky bugs,” *arXiv preprint arXiv:2404.10304*, 2024.
- [74] H. Taherkhani and H. Hemmati, “Valtest: Automated validation of language model generated test cases,” 2024. [Online]. Available: <https://arxiv.org/abs/2411.08254>
- [75] Z. Yuan *et al.*, “No more manual tests? evaluating and improving chatgpt for unit test generation,” *arXiv preprint arXiv:2305.04207*, 2023.

- [76] B. Kurtz *et al.*, “Mutant subsumption graphs,” in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2014, pp. 176–185.
- [77] P. Liu *et al.*, “Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing,” *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.
- [78] J. Zhang *et al.*, “Repairing bugs in python assignments using large language models,” *arXiv preprint arXiv:2209.14876*, 2022.
- [79] N. Nashid, M. Sintaha, and A. Mesbah, “Retrieval-based prompt selection for code-related few-shot learning,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2450–2462.
- [80] S. J. Kaufman *et al.*, “Prioritizing mutants to guide mutation testing,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1743–1754.
- [81] R. Just, B. Kurtz, and P. Ammann, “Inferring mutant utility from program context,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 284–294.
- [82] J. Zhang *et al.*, “Predictive mutation testing,” in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 342–353.
- [83] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in python,” *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.
- [84] R. G. Kurtz Jr, “Improving mutation testing with dominator mutants,” Ph.D. dissertation, George Mason University, 2018.
- [85] M. Barboni, A. Morichetta, and A. Polini, “Sumo: A mutation testing strategy for solidity smart contracts,” in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 2021, pp. 50–59.
- [86] S. Dou *et al.*, “What’s wrong with your code generated by large language models? an extensive study,” *arXiv preprint arXiv:2407.06153*, 2024.
- [87] S. B. Hossain *et al.*, “A deep dive into large language models for automated bug localization and repair,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1471–1493, 2024.