



UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

TESI DI LAUREA

Robustness-Driven Assertion Generation in Solidity Smart Contracts Using LLM Prompting Techniques

RELATORE

Prof. Dario Di Nucci

Dott. Gerardo Iuliano

Università degli Studi di Salerno

CANDIDATO

Daniele Carangelo

Matricola: 0522501696

Anno Accademico 2024-2025

Questa tesi è stata realizzata nel

sesa^{lab}
SOFTWARE ENGINEERING
SALERNO

“Computer Science is no more about computers than astronomy is about telescopes.”

- Edsger Wybe Dijkstra

Abstract

Smart contracts are self-executing programs deployed on blockchain platforms, commonly written in Solidity. They are used to automate and enforce agreements between parties without intermediaries, enabling applications such as decentralized finance (DeFi), token exchanges, and voting systems. Ensuring their correctness is critical, as bugs can lead to severe financial losses. Assertions play a key role in detecting logic errors and enforcing invariants during execution. However, manually writing meaningful assertions is error-prone and often neglected.

This thesis explores the automatic generation of assert statements in Solidity smart contracts using Large Language Models (LLMs). It evaluates different prompting techniques to guide the generation process and assesses the syntactic correctness and robustness of the generated assertions through formal verification using a solver. Three prompting strategies were employed: Chain-of-Thought, Chain-of-Thought comment-free, and Personas with Few-shot. The syntactic correctness of the generated assertions was evaluated using the solc compiler, while formal verification was performed through VeriSol and SMTChecker.

The results indicate that, for certain Solidity versions, particularly versions 0.6 and 0.7, as well as for large contracts, the model struggles to produce code that is both syntactically and formally correct. Regarding prompting techniques, the Chain-of-Thought approach, due to its explicit reasoning, proved more effective than the other methods considered.

Contents

List of Figures	iii
List of Tables	iv
1 Introduction	1
1.1 Application Context	1
1.2 Motivations and Objectives	2
1.3 Results Obtained	3
2 Background and State of the Art	4
2.1 Background	4
2.1.1 Blockchain, Smart Contracts, and their Invariants	4
2.1.2 Large Language Models and Prompt Engineering	6
2.2 Related Work	7
3 Methodology	12
3.1 Research Questions	12
3.2 Architecture	13
3.3 Proposed Method	13
3.3.1 Generation	14
3.3.2 Compilation	17

3.3.3	Formal Verification	17
4	Results	19
4.1	RQ ₀ : To what extent can LLM generate invariants for Solidity smart contracts?	19
4.2	RQ ₁ : How is the LLM capable of generating syntactically valid and compilable Solidity code?	22
4.3	RQ ₂ : Are the invariants generated by the LLM formally correct with respect to the contract's specifications?	24
4.4	Discussion of the Results	26
5	Threats to Validity	29
6	Conclusions	31
	Bibliography	32

List of Figures

3.1	Pipeline Architecture	13
4.1	Comparison of invariants per 100 LOC	20
4.2	Comparison of invariants per 100 SLOC	21
4.3	Bar Chart LOC-SLOC Average	21
4.4	Compiled contracts per Solidity version for each prompting method	22
4.5	Box Plot Contract size distribution	23
4.6	Box Plot Contract size distribution (No comments)	23

List of Tables

2.1	Comparison among analysis tools for invariant in Solidity contracts [1]	9
4.1	Generation results for different prompts	19
4.2	Compilation Results by Category for All Prompting Strategies	24
4.3	Verification Results for All Prompting Strategies in Solidity v0.4 - 0.7	25
4.4	Verification Results for All Prompting Strategies in Solidity v0.4 - 0.7	25
4.5	Verification Results for All Prompting Strategies on the Contracts in Solidity v0.8	26
4.6	Global Verification Results for All Prompting Strategies in Solidity v0.8	26

CHAPTER 1

Introduction

1.1 Application Context

Over the years, smart contracts have taken on a central role in the development of decentralized applications, establishing themselves as a fundamental component of the blockchain ecosystem and of decentralized finance (DeFi) protocols. Their autonomous and immutable nature makes them powerful tools, but at the same time exposes them to logical errors and vulnerabilities that can result in financial losses and significant consequences for both users and the platforms that provide decentralized services. For this reason, ensuring the reliability and robustness of smart contracts has become a top priority in both academic research and industry.

To address these challenges, the software engineering community places strong emphasis on verification and validation techniques aimed at improving smart-contract robustness. Assertion generation, in particular, provides a valuable mechanism for uncovering incorrect assumptions, identifying edge cases, and enforcing expected behaviors during static and dynamic analysis. Despite its benefits, manually designing appropriate assertions requires expertise, careful reasoning, and a deep understanding of contract semantics. At the same time, the emergence of large language models (LLMs) has introduced new opportunities for supporting and

automating the development of secure software. Modern LLMs demonstrate impressive capabilities in code generation, pattern recognition, and contextual reasoning, making them promising tools for assisting developers in tasks such as refactoring, documentation, vulnerability detection, and test creation. However, these models also present limitations. Their generative nature means they may occasionally produce plausible-sounding but incorrect information, phenomena commonly referred to as hallucinations. In the context of security-critical systems such as smart contracts, relying on such outputs without proper safeguards can introduce additional risks rather than mitigating them.

This thesis explores the use of LLM prompting techniques to guide the generation of assertions in Solidity smart contracts with the aim of enhancing their overall robustness. The goal is to assess how such models can support developers in identifying invariants and critical conditions, ultimately contributing to a more secure and reliable smart contract lifecycle.

1.2 Motivations and Objectives

Developers often overlook the explicit definition of invariants and assertions within their contracts. In many cases, these elements are added only partially or are formulated in ways that are formally incorrect or semantically ambiguous. This lack of rigor increases the chance of introducing bugs or security vulnerabilities that may only become visible after deployment, when the immutability of smart contracts makes fixing them difficult and expensive. Furthermore, the existing scientific literature offers limited support for automated assertion generation in Solidity. Current research typically relies on finetuned language models, specifically trained on smart contract datasets, or on complex systems that combine static analysis tools, symbolic execution engines, or RAG systems. While these approaches can be effective, they require significant computational resources, curated datasets, and updates for new vulnerabilities. In contrast, few studies have explored whether large language models, guided only by well-designed prompting strategies, can generate meaningful and formally correct invariants without further optimization.

This thesis aims to bridge this gap by systematically evaluating how different

prompting techniques influence the generation of robustness-oriented assertions in Solidity smart contracts. The primary goals of the work are:

1. Assessing different prompt strategies for invariant generation, examining their ability to produce syntactically valid output.
2. Verify whether the inferred invariants are formally correct using a solver.
3. Understanding the limitations of a prompt-only approach and highlighting cases in which LLM hallucinations or reasoning failures hinder the generation of correct invariants.

1.3 Results Obtained

Several significant results emerged from the analysis. First, the compilation rate proved to be notably low: out of a dataset of approximately 2,300 contracts, only 20% were successfully compiled after inserting the generated invariants. In addition, the LLM exhibited difficulties in handling large contracts and those belonging to Solidity versions 0.6 and 0.7.

The formal verification phase revealed similar issues for versions 0.6 and 0.7, whereas for the remaining versions, the rate of violated invariants ranged from approximately 10% to 27%, depending on the prompt and the version. Furthermore, the formal verification results highlight the superior performance of the Chain-of-Thought prompt, which proved more effective in generating robust invariants than the other prompting strategies evaluated.

CHAPTER 2

Background and State of the Art

This chapter is divided into two main sections: the first explores the concepts useful for understanding the work carried out in this thesis, while the second analyzes the works in the literature, the related results, and their limitations.

2.1 Background

The first part of this section examines the fundamental concepts related to blockchain, smart contracts, and their respective programming languages, with a particular focus on the introduction of the notion of invariants. The second part is focused on an overview of Large Language Models and the main prompting techniques.

2.1.1 Blockchain, Smart Contracts, and their Invariants

Blockchain. The concept of blockchain was first introduced by S. Nakamoto within the Bitcoin protocol as a distributed system for cryptocurrency exchange [2]. A blockchain is structured as a ledger composed of interconnected blocks, each containing information such as the set of transactions, the block's hash, the hash of the previous block, and additional metadata. Blockchains, including the one im-

plemented in Bitcoin, feature these characteristics [3]: (i) decentralization: there is no central server, as the network nodes are distributed worldwide; (ii) immutability: once added, a block cannot be altered without modifying the entire chain; (iii) transparency: all transactions are public and accessible to anyone; (iv) security: cryptographic techniques and consensus mechanisms, such as Proof of Work or Proof of Stake, make the blockchain highly resistant to attacks.

The main limitation of Bitcoin concerns the difficulty of executing complex programs on its blockchain, even though it has been shown to be Turing-complete [4]. This constraint led to the development of Ethereum, a blockchain that incorporates a Turing-complete programming language designed to enable the creation of smart contracts. These contracts allow developers to implement systems of any kind directly on the blockchain using only a few lines of code [5].

Smart Contracts. A smart contract is a program deployed and recorded on a blockchain, capable of autonomously executing operations in a decentralized and transparent manner, without the need for intermediaries. These contracts are immutable by design, and once deployed on the blockchain, they ensure high levels of security and reliability. Deployment of a smart contract is performed through a dedicated creation transaction, during which a unique address is assigned to the contract. This address can then be used to send requests and interact with the functions exposed by the contract [6].

Solidity. Solidity is a high-level, Turing-complete programming language used for the development of smart contracts. Its syntax, which resembles JavaScript, supports features such as inheritance, polymorphism, libraries, and complex data types. At present, Solidity is the primary language for writing smart contracts on the Ethereum blockchain and, more generally, on blockchains compatible with the Ethereum Virtual Machine (EVM) [6].

Smart contract Invariant Invariants within Solidity smart contracts are logical properties that must remain true throughout the execution of the contract at designated points in the code. They play a crucial role in ensuring the security and correctness

of the contract’s behavior. Among the main vulnerabilities that may affect smart contracts are, for example, integer overflow/underflow, reentrancy, and unsafe usage of `delegatecall`, as well as other types of attacks. In Solidity, invariants are typically expressed through two main constructs: `require` and `assert` [1, 7]. Listing 2.1 provides an example of a contract written in Solidity with some Invariants.

```

1 pragma solidity ^0.8.0;
2 contract SimpleBank {
3     mapping(address => uint256) public balances;
4
5     function withdraw(uint256 amount) public {
6         require(balances[msg.sender] >= amount, "Insufficient funds");
7
8         uint256 oldBalance = balances[msg.sender];
9         balances[msg.sender] -= amount;
10
11         assert(balances[msg.sender] == oldBalance - amount);
12         payable(msg.sender).transfer(amount);
13     }
14 }

```

Listing 2.1: Example fo a Solidity smart contract

In the presented contract, two invariants are expressed through the **require** and **assert** constructs. The `require` statement in the `withdraw` function verifies a precondition, namely that the caller has sufficient funds to perform the withdrawal. The `assert`, on the other hand, checks an internal invariant of the contract and ensures that, after the withdrawal operation, the resulting balance is consistent with the expected change.

2.1.2 Large Language Models and Prompt Engineering

Large Language Models. A Language Model (LM) is a system equipped with advanced Natural Language Processing (NLP) capabilities, able to understand and generate text by predicting the probability of subsequent words based on the preceding context. Large Language Models (LLMs), such as GPT or LLaMA, are an advanced class of LMs characterized by a very large number of parameters which,

through the use of the Transformer architecture, enable efficient handling of sequential data and the detection of long-range dependencies within text. A key property of these models is in-context learning, which allows the system to learn from the given prompt and generate coherent and contextually relevant responses [8, 9].

Prompt Engineering. Prompt engineering was developed to enhance the capabilities of Large Language Models (LLMs) by introducing various techniques and strategies for crafting effective prompts without modifying the model’s parameters through fine-tuning [10]. The most commonly used prompting techniques include:

Zero-Shot Prompting. The prompt is designed to guide the model to perform the task without additional data or input examples. In this case, the model’s output relies solely on its prior knowledge.

Few-Shot Prompting. The prompt contains a small number of illustrative examples that help the model understand and execute the task. This approach significantly improves performance, although it increases the number of tokens required within the context window.

Chain-of-Thought (CoT). This technique encourages the model to explicitly articulate the logical steps or reasoning leading to the final answer, thereby enhancing accuracy, particularly in mathematical or logical tasks.

2.2 Related Work

This section reviews several works in the literature that employ Large Language Models to analyze or generate Solidity code, for example vulnerability detection or invariant generation.

Chachar et al. [11] investigated the ability of Large Language Models (LLMs) to identify vulnerabilities in smart contracts. Their study focused on three vulnerability types: Reentrancy, Timestamp Manipulation, and Unchecked External Call, selected as they were found to be among the most common according to a literature survey [12] conducted between 2018 and 2023.

The experiment involved three LLMs: GPT-4o, Mistral-7B, and Claude 3.5 Sonnet, and employed the In-Context Learning (ICL) prompting technique. Through ICL, the authors enriched the models with additional knowledge by embedding examples of vulnerable contracts directly into a prompt with this structure: (i) “Task Description” instructing the model to detect a specific vulnerability, (ii) “Vulnerability Description” providing a natural language description of the vulnerability, (ii) “Examples” smart contracts containing the target vulnerability.

The smart contract examples were obtained from datasets used in SmartBugs [13] and later manually labeled for each vulnerability type. Model performance was assessed using the F1-score, which was also used to compare the results from two static analysis tools: Conkas and Slither. GPT-4o outperformed the other models, achieving about 89% F1 on Reentrancy and scores between 66% and 69% for the other two vulnerabilities.

An additional comparison was performed with FELL MVP [14], a framework for smart contract vulnerability classification using LLMs. FELL MVP relies on a fine-tuning approach targeting a single vulnerability at a time and achieves approximately 95% F1 for Reentrancy .

The main limitation of both Chachar et al. [11] and FELL MVP [14] is their specialization in detecting only one vulnerability at a time. In the former case, the prompt explicitly defines the target vulnerability and provides examples exclusively related to it. In the latter, the model is fine-tuned on a single vulnerability class, limiting its ability to generalize.

The increasing adoption of LLM-based methods for automated code generation has raised the need to assess the capability of these models to produce secure and reliable software. Although several benchmarks have been proposed to evaluate LLM performance, most of them focus on mainstream languages such as Python and Java. SolEval [15] aims to address this gap by introducing a Solidity-specific benchmark that evaluates 10 LLMs across four key metrics: functional correctness, compilability, gas fee efficiency, and vulnerability rate. To compute the vulnerability rate, Slither [16] is used to analyze the generated code, taking into account only high-risk vulnerabilities with a high confidence score. Benchmark results show that the vulnerability rate ranges from 10.59% (DeepSeek-R1-Distill-Qwen) to 34% (GPT-

4o-mini), highlighting the presence of significant security weaknesses in Solidity code generated by LLMs .

Previously, several tools for Solidity code analysis have been explored with the aim of detecting implementation bugs within smart contracts. However, a new category of vulnerabilities, referred to as machine un-auditable bugs, has recently been identified. These bugs are related to the transaction context and cannot be detected by traditional analysis tools. In this context, SmartInv [1] has been proposed as a framework for the detection of machine un-auditable bugs and the inference of invariants within smart contracts. At its core, the framework employs a large language model (LLM) fine-tuned using a novel prompting strategy called Tier of Thought (ToT). This technique guides the model to reason across multiple levels. Specifically, the reasoning process is divided into three main stages: (i) identification of the contract’s transaction context, (ii) detection of critical points, and (iii) generation of invariants corresponding to the previously identified critical points. After invariant generation, the contracts are analyzed using two formal verification tools, Boogie and Corral, to assess their correctness. An ablation study was also conducted to evaluate the impact of the Tier of Thought (ToT) strategy within SmartInv. When this prompting method was removed, both accuracy and F1-score dropped by approximately 66%, thereby demonstrating the crucial role of ToT in the framework’s performance. Finally, SmartInv achieved the best results in invariant inference, producing fewer false positives compared to other analysis tools such as InvCon [17] and VeriSmart[18]. See Table 2.1 for more details.

Table 2.1: Comparison among analysis tools for invariant in Solidity contracts [1]

Tool	LOC (avg)	Invariants/Contract	FP/Contract
SmartInv	1,621	6.00	0.32
InvCon	862	11.70	2.41
VeriSmart	354	3.00	0.92

SmartInv was also able to detect 119 zero-day bugs in smart contracts already deployed on the blockchain, some of which were classified as high-severity vulnerabilities. Among the main limitations of the framework is the restricted input token

length of the underlying language model (LLM), which prevents the direct analysis of large-scale contracts. Specifically, for contracts exceeding 2,000 lines of code (LOC), a summarization procedure was applied, retaining only the most relevant functions. However, this approach leads to a loss of analysis precision, as the framework does not have access to the entire contract context, potentially affecting the completeness and reliability of the verification process.

Liu et al. proposed a property generation tool named PropertyGPT, which is based on Large Language Models (LLMs) and ensures that the newly generated properties are compilable, appropriate, and verifiable. A key distinction from comparable models like SmartInv is that PropertyGPT was not fine-tuned; instead, it utilizes an in-context learning technique known as Retrieval-Augmented Generation (RAG). The property generation process is structured in distinct phases:

Phase 1: Embedding and Knowledge Base Construction: The first stage involves embedding, where properties and corresponding critical code, primarily sourced from expert-written audit reports (e.g., Certora reports), are stored in a vector database. Notably, the reference properties themselves are generally not embedded, as the code is used as the search key.

Phase 2: Retrieval: This phase retrieves properties and code from the vector database that are similar to the target smart contract or function provided as input to PropertyGPT. Retrieval is performed by querying the vector database, typically using a metric such as cosine similarity on the embedded vectors.

Phase 3: Generation (In-Context Learning): The LLM receives the code under testing, a generation prompt, and the relevant reference properties retrieved in the preceding phase, enabling it to generate a candidate property.

Phase 4: Refinement (Feedback & Revise): Following generation, the properties enter an iterative Feedback & Revise loop. This process uses feedback from the compiler as an external oracle to guide the LLM in iteratively correcting and revising the properties to ensure their syntactic and grammatical correctness.

Final Phase (Verification): In the final step, the compilable properties are passed to a dedicated prover for formal verification, aiming to discover smart contract

vulnerabilities.

PropertyGPT demonstrated superior performance in vulnerability detection, detecting several CVEs and surpassing tools such as Mythril, Slither, and Manticore [19, 16, 20]. Furthermore, it successfully uncovered 12 zero-day vulnerabilities in real-world projects. The main limitation of this work concerns the limited knowledge provided to the RAG system, which is based solely on a fixed collection of Certora reports. This necessitates the continuous updating of the database to ensure adequate coverage of emerging and recent vulnerabilities [21].

CHAPTER 3

Methodology

The objective of this thesis is to analyze the ability of large language models (LLMs) to generate invariants for Solidity smart contracts, examining the impact of different prompting techniques. Furthermore, the thesis aims to assess the correctness of the generated invariants, both from a syntactic and a formal point of view. An online appendix containing all datasets and scripts used in this work is available at <https://github.com/daniele-carangelo/InvariantGen>.

3.1 Research Questions

This research aims to answer the following research questions:

RQ₀: To what extent can LLMs generate invariants for Solidity smart contracts?

This research question examines how an LLM behaves when generating invariants and to what extent, to obtain an overall understanding of the generation phase.

RQ₁: How is the LLM capable of generating syntactically valid and compilable Solidity code?

Invariant generation is meaningful only if the produced assertions can be correctly

integrated into a smart contract without altering its syntactic validity. This question evaluates the syntactic robustness and practical usability of the generated code.

RQ₂: Are the invariants generated by the LLM formally correct with respect to the contract’s specifications?

Beyond syntactic correctness, invariants must accurately represent properties that hold across all possible contract executions. This research question, therefore, focuses on the semantic and formal correctness of the generated invariants.

3.2 Architecture

To address these research questions, the following method was used, structured into three main phases as illustrated in the figure 3.1:

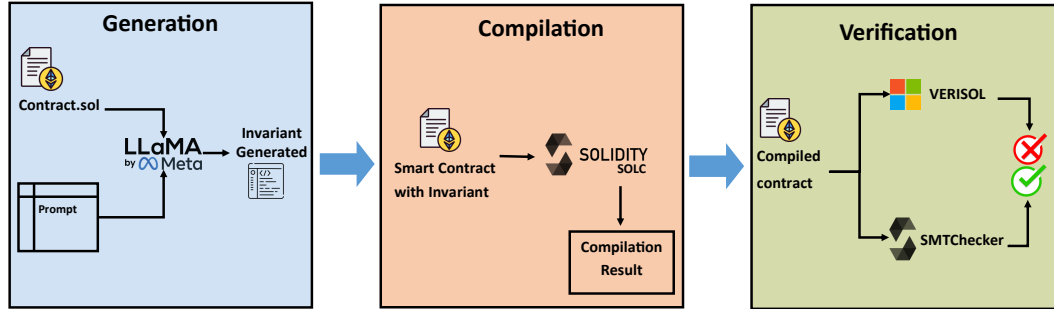


Figure 3.1: Pipeline Architecture

3.3 Proposed Method

The proposed method is structured into three main phases, presented in their respective sections: Generation, Compilation, and Verification. The Generation phase outlines the tools and strategies employed for invariant inference. The Compilation phase describes the process used to assess the syntactic correctness of the generated outputs. Finally, the Verification section presents the procedure through which the contracts were formally verified.

3.3.1 Generation

The following paragraphs describe the selected model along with its corresponding hyperparameters, the hardware employed, and the complete process of prompt engineering and data preparation.

Used LLM and Hardware. The model selected for the generation phase is LLaMA 3.3 70B, an open-source large language model. To avoid the costs associated with API access to online models, all generation tasks were performed locally. The input dataset includes approximately 2,300 smart contracts, and conducting multiple generations while testing different prompts and parameters would have resulted in substantial expenses if executed through cloud-based services.

The model was run using the LM Studio environment on a workstation equipped with an NVIDIA GeForce RTX 4090 GPU. To fit within the memory constraints of the available hardware, the LLM was employed in a 3-bit quantized version.

Hyperparameter Configuration. Among the most important parameters for an LLM we have: **temperature** regulates the level of randomness in token selection, the **context window size** determines the maximum amount of information the model can consider at once, while **Top-K** and **Top-P** sampling control the probability distribution of candidate tokens. The following parameters were configured for the generation process: temperature = 0.7, context window size = 8000, Top K = 40, and Top P = 0.95. The parameters used are largely standard; the only substantial modification concerns the context window size, which was increased to allow the model to process a greater number of contracts.

Prompt Engineering. The prompt engineering phase primarily focused on designing two prompts that guided the invariant generation process. Specifically, the first prompt was developed using the Chain-of-Thought (CoT) technique, thereby enabling the model’s reasoning capabilities. The initial version of the prompt was created manually and subsequently refined using OpenAI’s prompt optimizer¹, which

¹<https://platform.openai.com/chat/edit?models=gpt-5&optimize=true>

enhanced its structure according to established prompt engineering best practices. The resulting final prompt is shown below:

```

1 You are given a Solidity smart contract. First, meticulously reason step
  by step and identify possible invariants (such as supply conservation,
  non-negative balances, or ownership conditions) using detailed,
  silent chain-of-thought reasoning. Do not display your reasoning. Next
  , give me only the invariant statements to be inserted at the critical
  program points and in which line. no explanation or extra text.
2
3 Output format: line_number: invariant_statement;
4 example: 20: assert(funcA2(funcA1())==12);

```

Listing 3.1: Chain-of-Thought (CoT) prompt

The second prompt adopts a hybrid approach combining persona-based prompting and few-shot prompting. Through the persona specification, the model was assigned the role of a smart contract security expert, while the few-shot component enabled in-context learning by providing two examples along with their expected outputs. The resulting prompt from this phase is as follows:

```

1 You are a Solidity Smart Contract Security Auditor and Formal
  Verification expert. Your primary task is to **analyze the provided
  Solidity code for its business logic, critical state variables, and
  security assumptions**. Based on this deep understanding, you must **
  identify and generate essential security invariants**.
2
3 Example 1:
4 1:  contract SimpleToken {
5 2:      mapping(address => uint256) public balances;
6 3:      uint256 public totalSupply;
7 4:
8 5:      function transfer(address to, uint256 amount) public {
9 6:          require(balances[msg.sender] >= amount);
10 7:          balances[msg.sender] -= amount;
11 8:          balances[to] += amount;
12 9:      }
13 10:
14 11:     function mint(address to, uint256 amount) public {
15 12:         balances[to] += amount;

```

```

16 13:         totalSupply += amount;
17 14:     }
18 15: }
19
20 Expected Output:
21 9: assert(balances[msg.sender] + balances[to] == balances[msg.sender] +
    amount);
22 14: assert(totalSupply >= balances[to]);
23
24 Example 2:
25 1:  contract Voting {
26 2:      mapping(address => bool) public hasVoted;
27 3:      mapping(uint256 => uint256) public votes;
28 4:      uint256 public totalVotes;
29 5:
30 6:      function vote(uint256 proposalId) public {
31 7:          require(!hasVoted[msg.sender]);
32 8:          hasVoted[msg.sender] = true;
33 9:          votes[proposalId]++;
34 10:         totalVotes++;
35 11:     }
36 12: }
37
38 Expected Output:
39 9: assert(hasVoted[msg.sender] == true);
40 11: assert(totalVotes > 0);
41 12: assert(votes[proposalId] <= totalVotes);
42
43 Output format: line_number: invariant_statement;
44 example: 20: assert(funcA2(funcA1())==12);
45 Do NOT include explanations, comments, or additional text.
46 Only output the line numbers and assert statements.

```

Listing 3.2: Personas with Few-shot prompt

In both prompts, the structure of the expected output was explicitly defined in order to facilitate the parsing operations performed after generation. A third generation was performed using the previously described Chain-of-Thought prompt, after removing all comments from the contracts.

Data Preparation. With respect to data preparation, the initial dataset did not require substantial modifications, as it was already organized by Solidity version and the contracts were properly formatted. Prior to generation, the lines of code of each contract were numbered to avoid ambiguity and ensure an accurate mapping between each generated invariant and its intended insertion point. To complete the proposed generation without comments, a duplicated version of the dataset was also created, from which all comments within the contracts were removed. After the generation phase, the produced invariants were inserted into the original contracts at the specified positions, and the line numbering was subsequently removed.

3.3.2 Compilation

As a result of the previous phase, a dataset was obtained consisting of the original contracts containing the generated invariants inserted in their corresponding positions. To support RQ1, and thus assess whether both the generated code and its insertion points were syntactically valid, the contracts were compiled using the Solidity compiler *solc*. Since the dataset includes contracts written in different versions of the language, the selection of the appropriate compiler was automated using the *solc-select* tool. After the compilation process, all contracts that successfully compiled were collected into a new set of valid contracts. At this stage, the correctness of individual invariants was not evaluated; instead, the focus was on assessing their broader impact on the syntactic validity of the contract as a whole.

3.3.3 Formal Verification

The final phase of the pipeline concerns the formal verification of the generated assertions and is carried out exclusively on the contracts that successfully compiled in the previous step. Formal verification is performed using tools that translate the Solidity contract into a mathematical model and then rely on a solver to explore the input space in search of values that may violate the specified conditions. Two tools were used in this experiment: VeriSol and SMTChecker. VeriSol supports Solidity versions 0.4-0.7, whereas SMTChecker is compatible with more recent versions of the language and was therefore used to verify contracts starting with version 0.8.

VeriSol², developed by Microsoft, takes a Solidity contract as input, translates it into an intermediate language called Boogie, and subsequently performs formal verification using a solver. SMTChecker³, on the other hand, is integrated into the Solidity compiler and can be enabled via a command-line flag. Both tools rely on the same underlying solver, Z3⁴.

Each verified contract was assigned to one of three categories: valid, invalid, or not verifiable. A contract was labeled valid when all its invariants were successfully verified and no counterexample was found that could falsify any of them. The invalid category was used when at least one invariant was violated during the verification process. Finally, a contract was classified as not verifiable when the verification procedure could not be completed due to external issues, such as tool limitations, translation errors, or solver timeouts.

²<https://www.microsoft.com/en-us/research/project/verisol-a-formal-verifier-for-solidity-based-smart-contracts/?locale=fr-ca>

³<https://docs.soliditylang.org/en/latest/smtchecker.html>

⁴<https://github.com/Z3Prover/z3>

CHAPTER 4

Results

This chapter presents the results and answers to the research questions.

4.1 RQ₀: To what extent can LLM generate invariants for Solidity smart contracts?

Table 4.1 reports the results for each prompt under evaluation. The “# Contracts” column indicates the number of contracts contained in the initial dataset. The “# Contracts with Invariants” column includes the contracts for which the LLM produced invariants that adhered to the required output format. Finally, the “# Contracts w/o Invariants” column contains all cases where generation failed, including errors such as exceeding the context window or producing output in a non-parsable format.

Table 4.1: Generation results for different prompts

Prompting Method	# Contracts	# Contracts with Invariants	# Contracts w/o Invariants
Chain-of-Thought	2,285	1,446	839
CoT (no comment)		2,227	58
Personas & Few-shot		1,688	597

4.1 – RQ_0 : To what extent can LLM generate invariants for Solidity smart contracts?

Figure 4.1 presents the ratio of generated invariants per 100 LOC, grouped by Solidity version and prompt type.

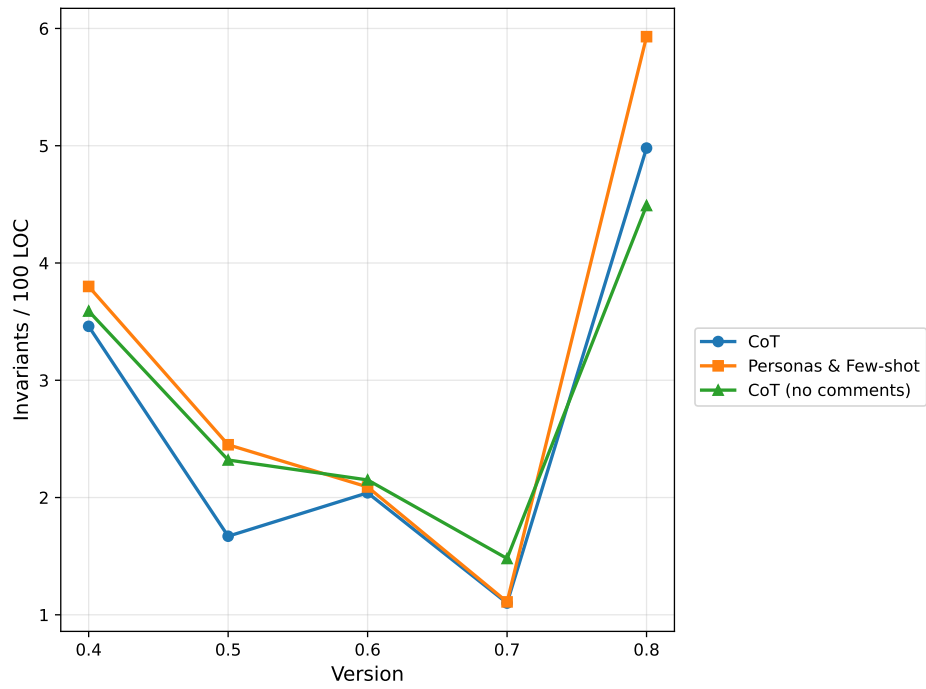


Figure 4.1: Comparison of invariants per 100 LOC

Figure 4.2 presents the ratio of generated invariants per 100 SLOC, grouped by Solidity version and prompt type.

4.1 – RQ_0 : To what extent can LLM generate invariants for Solidity smart contracts?

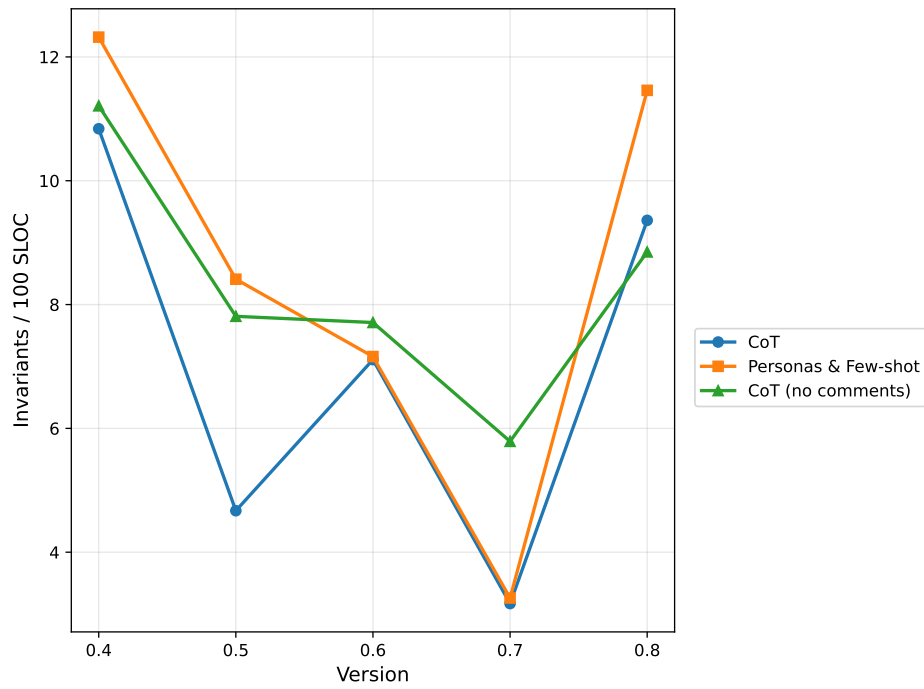


Figure 4.2: Comparison of invariants per 100 SLOC

The chart in Figure 4.3 shows the average LOC and SLOC per contract, grouped by Solidity version. This view helps us understand the number of comments present in the code.

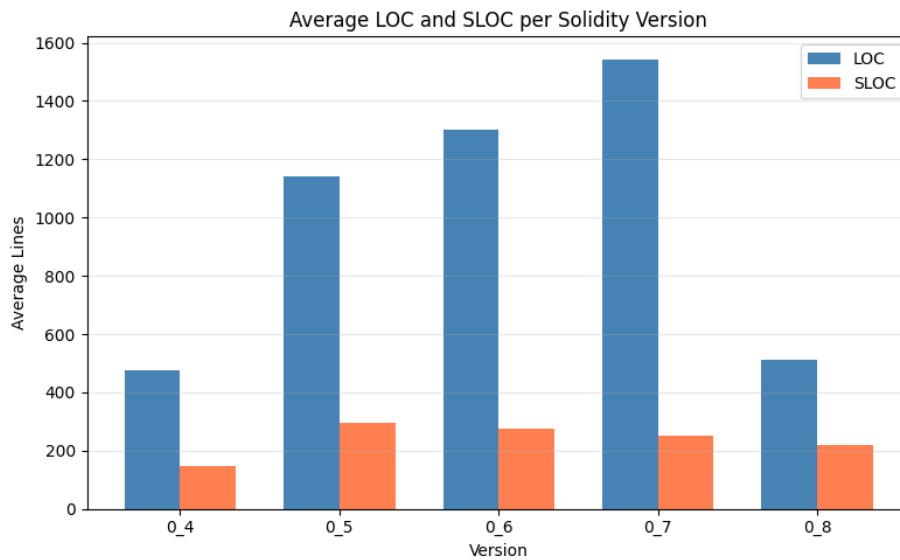


Figure 4.3: Bar Chart LOC-SLOC Average

4.2 – RQ₁: How is the LLM capable of generating syntactically valid and compilable Solidity code?

RQ₀ Summary: The Solidity version and the prompting technique used had an impact on the number of invariants generated, whereas contract size negatively affected the rate of successful generations.

4.2 RQ₁: How is the LLM capable of generating syntactically valid and compilable Solidity code?

To address this research question, the analysis focused on the compilation results of the contracts with the generated invariants added. Figure 4.4 presents the compilation results for the three generations performed, organized by Solidity version and prompt method.

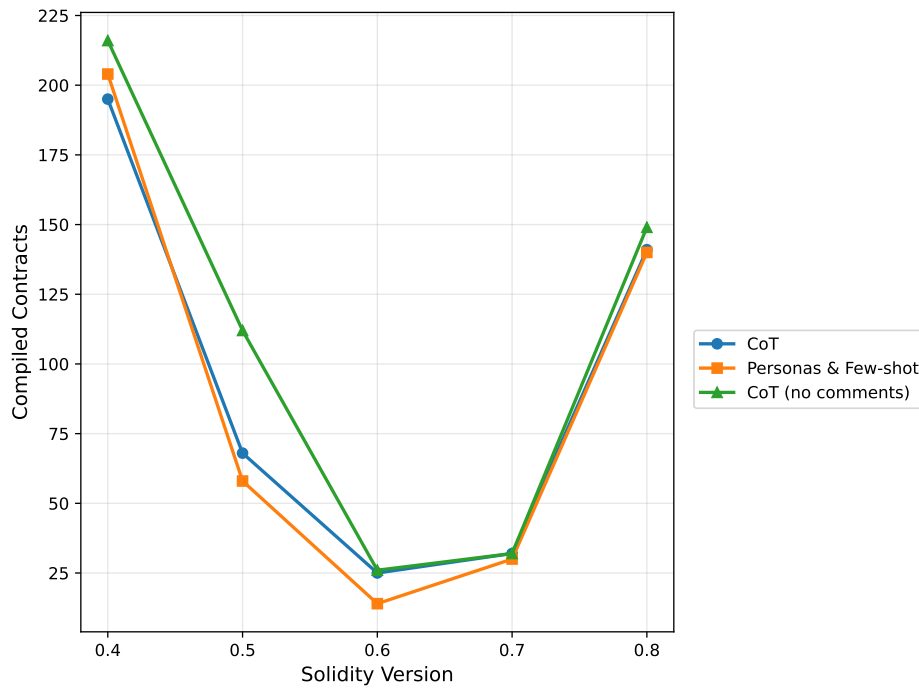


Figure 4.4: Compiled contracts per Solidity version for each prompting method

The box plot in Figure 4.5 shows the distribution of contract sizes.

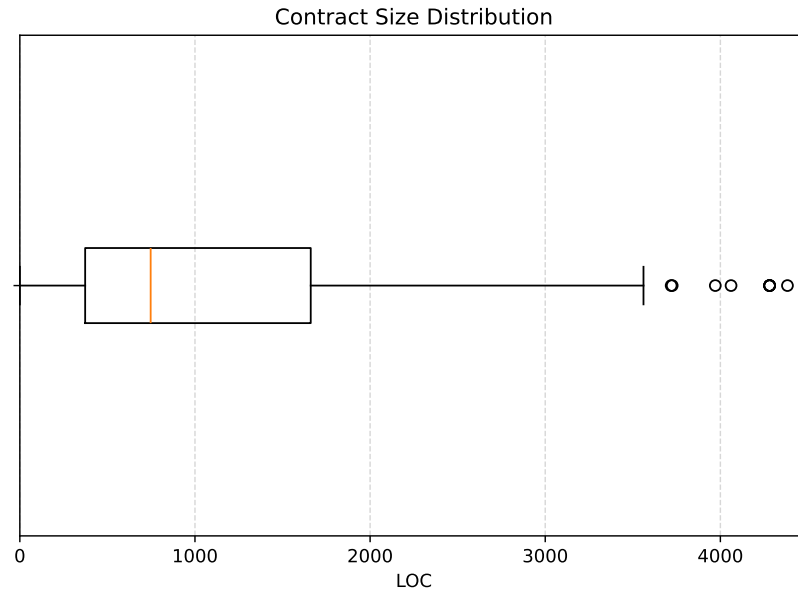


Figure 4.5: Box Plot Contract size distribution

The box plot in Figure 4.6 shows the distribution of contract sizes excluding comments.

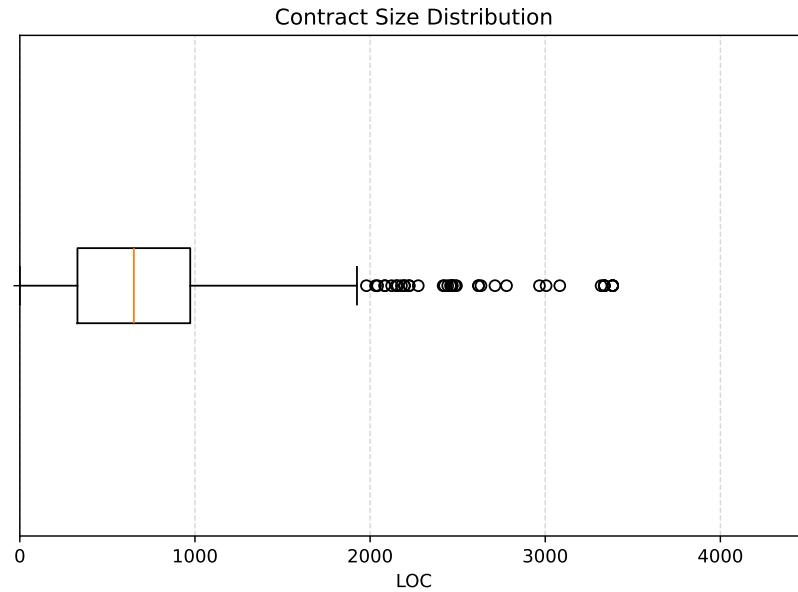


Figure 4.6: Box Plot Contract size distribution (No comments)

Table 4.2 presents the number of compiled contracts categorized into three size groups: small, medium, and large, and divided by prompt.

4.3 – *RQ₂: Are the invariants generated by the LLM formally correct with respect to the contract’s specifications?*

Table 4.2: Compilation Results by Category for All Prompting Strategies

Prompt	Small ($\leq Q1$)	Medium (Q1–Q3)	Large ($> Q3$)
CoT	290	171	0
Personas & Few-shot	312	134	0
CoT (no Comment)	295	218	21

RQ₁ Summary: The Solidity versions used in the contracts, together with their size, affect the syntactic correctness of the generated invariants.

4.3 **RQ₂: Are the invariants generated by the LLM formally correct with respect to the contract’s specifications?**

To address this research question, the formal verification results obtained from the successfully compiled contracts were analyzed. The results presented in the following sections were collected after running both VeriSol and SMTChecker.

Table 4.3 presents the results of running VeriSol on contracts written in Solidity versions ranging from 0.4 to 0.7. For each version and prompt, the tables report the number of contracts successfully verified by VeriSol, the number of generated invariants, the number of violated invariants, and the corresponding percentage of invariants that were found to be invalid.

4.3 – RQ_2 : Are the invariants generated by the LLM formally correct with respect to the contract’s specifications?

Table 4.3: Verification Results for All Prompting Strategies in Solidity v0.4 - 0.7

CoT Prompt				
Version	Contracts	Inv. Generated	Inv. Violated	% Violated
0.4	45	420	89	21.19%
0.5	34	324	31	9.57%
0.6	1	8	0	0.00%
0.7	3	26	5	19.23%
Personas & Few-shot Prompt				
0.4	44	500	139	27.80%
0.5	39	353	58	16.43%
0.6	1	10	0	0.00%
0.7	2	13	7	53.85%
CoT Prompt (No Comments)				
0.4	48	399	83	20.80%
0.5	47	448	47	10.49%
0.6	1	7	0	0.00%
0.7	2	8	1	12.50%

Table 4.4 presents the global verification results. Specifically, they report, for each prompt, the total number of contracts analyzed by VeriSol, the number of contracts with at least one violated invariant, the number of contracts with no violated invariants, and finally, the number of contracts that could not be verified due to limitations or errors related to VeriSol.

Table 4.4: Verification Results for All Prompting Strategies in Solidity v0.4 - 0.7

Metric	CoT	Personas & Few-shot	CoT (No Comment)
Contracts	83	86	98
Violated Contracts	51 - 61.45%	54 - 62.79%	65 - 66.33%
Safe Contracts	9 - 10.84%	5 - 5.81%	8 - 8.16%
Non-verifiable	23 - 27.71%	27 - 31.40%	24 - 24.49%

For the contracts written in Solidity version 0.8, the verification results were obtained by running SMTChecker, the formal analysis tool integrated into the solc

compiler. Table 4.5 presents the results of the execution. For each prompt, the tables report the number of generated invariants, the number of violated invariants, and the corresponding percentage of invariants that were found to be invalid.

Table 4.5: Verification Results for All Prompting Strategies on the Contracts in Solidity v0.8

Metric	CoT	Personas & Few-shot	CoT (No Comment)
Inv. Generated	1438	1445	1297
Inv. Violated	237	360	210
% Violated	16.48%	24.91%	16.19%

Table 4.6 presents the global verification results. Specifically, they report, for each prompt, the total number of contracts analyzed by SMTChecker, the number of contracts with at least one violated invariant, the number of contracts with no violated invariants, and finally, the number of contracts that could not be verified due to limitations or errors related to SMTChecker.

Table 4.6: Global Verification Results for All Prompting Strategies in Solidity v0.8

Metric	CoT	Personas & Few-shot	CoT (No Comment)
Analyzed Contracts	141	140	149
Contracts Violated	78 - 55.32%	119 - 85%	107 - 71.81%
Safe Contracts	45 - 31.91%	12 - 8.57%	28 - 18.79%
Non-verifiable	18 - 12.77%	9 - 6.43	14 - 9.40%

RQ₂ Summary: The Solidity versions used in the contracts, together with the prompting technique adopted for the generation process, may affect the formal correctness of the produced invariants, with reasoning-based prompts showing a particular advantage.

4.4 Discussion of the Results

Within the context of RQ1, we need to observe the limited number of contracts successfully compiled for Solidity versions 0.6 and 0.7. As shown in Figure 4.3, these

versions contain a particularly high number of comments within the code, which could potentially affect invariant generation and, consequently, the compilation process. To verify this hypothesis, the generation phase was repeated using the input contract dataset with all comments removed. The results, presented in Figure 4.4, refute this hypothesis, indicating that comments do not influence the generation of syntactically correct code.

The lifecycle of Solidity versions may have affected the amount of training data available to the used LLM. An analysis of the release dates shows that versions 0.6 and 0.7 remained active for approximately six months, whereas version 0.8 has been active for about five years, version 0.5 for one year and four months, and version 0.4 for roughly two years. Therefore, the shorter lifecycle of versions 0.6 and 0.7 is likely to have negatively influenced the quality of the generated outputs.

Contract size affects the results: no contract classified as Large was successfully compiled. This outcome may be due to limitations of the LLM when handling inputs requiring a very large context window. Overall, the compilation rate is quite low, approximately 20%. Moreover, an analysis of the compiler reports reveals that most errors concern the placement of the invariant rather than the invariant itself.

Analyzing the results of RQ2, the issue observed previously in Solidity versions 0.6 and 0.7 resurfaces. Since the verification phase was performed only on successfully compiled contracts, and these versions already exhibited a limited number of compilations, the number of translated and verified contracts decreased even further. This reduction makes the results not comparable to those of the other versions.

The type of prompt used also affects the results. While the different prompts performed similarly during the compilation phase, a clear distinction emerges in the verification phase: the Chain-of-Thought (CoT) prompt outperforms the hybrid Personas and few-shot approach. Despite being structurally simpler, enabling explicit reasoning allowed the model to generate more robust invariants.

An additional issue encountered during the verification phase concerns the VeriSol tool. Specifically, translating contracts into the Boogie intermediate language produced several errors, preventing both the translation and the subsequent verification of multiple contracts. This problem stems from the lack of a complete mapping between certain Solidity constructs and their Boogie counterparts, which caused the

translation process to fail.

Concerning the generation phase, it was observed that the model expressed invariants exclusively through “assert” statements, neglecting the use of constructs such as “require” to represent pre- and post-conditions. This behavior may be attributable to the examples included in the prompt or to the output format specification, which explicitly referred only to assert statements.

CHAPTER 5

Threats to Validity

This section discusses the potential threats to the validity of the study and the strategies adopted to mitigate them [22].

Internal Validity. A threat to internal validity stems from the rapidly evolving nature of the Solidity language. Different versions introduce features, security mechanisms, and syntax changes, which may influence the LLM’s understanding. For example, Solidity 0.6 had a lifespan of only seven months, limiting the amount of publicly available training data for that version. Consequently, the LLM’s familiarity with certain versions may vary significantly [23].

External Validity. External validity threats relate to the characteristics of the dataset used to evaluate the model. The source and nature of the smart contracts could potentially affect the results, especially if the dataset contained artificial or non-representative examples. To reduce this threat, all contracts included in the study were extracted from the Ethereum blockchain [5], mined backward in time from May 2025. As such, they represent real-world, actively deployed, and practically used smart contracts, increasing the likelihood that the dataset reflects authentic development practices and real operational conditions.

Another external threat concerns the exclusive use of a single Large Language Model. Although this may limit the breadth of the evaluation, the selected model is open-source and widely adopted within the research community, ensuring transparency, reproducibility, and accessibility of the methodology. Its open-source nature also allows inspection of model behavior and weights, partially mitigating concerns related to model-specific biases and opacity.

Construct Validity. A construct validity threat concerns the use of two distinct formal verification tools, VeriSol and SMTChecker, to evaluate the correctness of the generated invariants. Since the two tools adopt different verification pipelines and translation mechanisms, their internal differences could potentially influence the outcomes reported in RQ2. However, this threat is substantially mitigated because both tools rely on the same underlying SMT solver, Z3 [24].

CHAPTER 6

Conclusions

The objective of this thesis is to evaluate the ability of a LLM to generate invariants for Solidity smart contracts. The generation process relied on the LLaMA 3.3 model, supported by a dataset of contracts mined from the Ethereum blockchain [5]. Three prompting strategies were employed: Chain-of-Thought (CoT), a comment-free variant of CoT, and Personas with few-shot examples [10]. The evaluation focused on two main aspects: the syntactic correctness of the generated invariants, verified using the Solidity compiler solc, and their formal correctness, assessed through the VeriSol and SMTChecker tools. Throughout the experiment, additional factors that might influence invariant generation were examined, including the Solidity version, contract size, and the presence of comments. The results show that both the Solidity version and the contract size can significantly affect the correctness of the inferred invariants. Moreover, the choice of prompt proved critical, underscoring the importance of prompt engineering: the Chain-of-Thought prompt achieved superior performance compared to the more elaborate Personas in a few-shot configuration. This work evaluated the feasibility of inferring invariants for Solidity smart contracts using only LLMs and prompting techniques, demonstrating that these methods alone are not yet sufficient without support from additional analysis tools or manual review. Nonetheless, the findings open promising avenues for future research in this area.

Bibliography

- [1] S. J. Wang, K. Pei, and J. Yang, “Smartinv: Multimodal learning for smart contract invariant inference,” in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 2217–2235. (Citato alle pagine iv, 6 e 9)
- [2] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” *Available at SSRN 3440802*, 2008. (Citato a pagina 4)
- [3] M. Di Pierro, “What is the blockchain?” *Computing in Science & Engineering*, vol. 19, no. 5, pp. 92–95, 2017. (Citato a pagina 5)
- [4] C. S. Wright, “A proof of turing completeness in bitcoin script,” in *Proceedings of SAI Intelligent Systems Conference*. Springer, 2019, pp. 299–313. (Citato a pagina 5)
- [5] V. Buterin *et al.*, “A next-generation smart contract and decentralized application platform,” *white paper*, vol. 3, no. 37, pp. 2–1, 2014. (Citato alle pagine 5, 29 e 31)
- [6] M. Wohrer and U. Zdun, “Smart contracts: security patterns in the ethereum ecosystem and solidity,” in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 2–8. (Citato a pagina 5)
- [7] Z. Chen, Y. Liu, S. M. Beillahi, Y. Li, and F. Long, “Demystifying invariant effectiveness for securing smart contracts,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1772–1795, 2024. (Citato a pagina 6)

-
- [8] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, “A survey on evaluation of large language models,” *ACM transactions on intelligent systems and technology*, vol. 15, no. 3, pp. 1–45, 2024. (Citato a pagina 7)
- [9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017. (Citato a pagina 7)
- [10] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, “A systematic survey of prompt engineering in large language models: Techniques and applications,” *arXiv preprint arXiv:2402.07927*, 2024. (Citato alle pagine 7 e 31)
- [11] B. Chachar, M. Cavazza, A. Bracciali, P. Ferrara, and A. Cortesi, “Finding vulnerabilities in solidity smart contracts with in-context learning,” in *ITASEC/SERICS*, 2025. [Online]. Available: <https://api.semanticscholar.org/CorpusID:278885653> (Citato alle pagine 7 e 8)
- [12] C. De Baets, B. Suleiman, A. Chitizadeh, and I. Razzak, “Vulnerability detection in smart contracts: A comprehensive survey,” *arXiv preprint arXiv:2407.07922*, 2024. (Citato a pagina 7)
- [13] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, “Smartbugs: A framework to analyze solidity smart contracts,” in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 2020, pp. 1349–1352. (Citato a pagina 8)
- [14] Y. Luo, W. Xu, K. Andersson, M. S. Hossain, and D. Xu, “Fellmvp: An ensemble llm framework for classifying smart contract vulnerabilities,” in *2024 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2024, pp. 89–96. (Citato a pagina 8)
- [15] Z. Peng, X. Yin, R. Qian, P. Lin, Y. Liu, H. Zhang, C. Ying, and Y. Luo, “Sol-eval: Benchmarking large language models for repository-level solidity code generation,” *arXiv preprint arXiv:2502.18793*, 2025. (Citato a pagina 8)

-
- [16] 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. (Citato alle pagine 8 e 11)
- [17] Y. Liu and Y. Li, "Invcon: A dynamic invariant detector for ethereum smart contracts," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–4. (Citato a pagina 9)
- [18] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1678–1694. (Citato a pagina 9)
- [19] Mythril. (2025) Mythril. [Online]. Available: <https://github.com/ConsenSysDiligence/mythril> (Citato a pagina 11)
- [20] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189. (Citato a pagina 11)
- [21] Y. Liu, Y. Xue, D. Wu, Y. Sun, Y. Li, M. Shi, and Y. Liu, "Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation," *arXiv preprint arXiv:2405.02580*, 2024. (Citato a pagina 11)
- [22] K. Anglin, Q. Liu, and V. C. Wong, "A primer on the validity typology and threats to validity in education research," *Asia Pacific Education Review*, vol. 25, no. 3, pp. 557–574, 2024. (Citato a pagina 29)
- [23] Solidity. (2025) release version. [Online]. Available: <https://www.soliditylang.org/blog/category/releases/> (Citato a pagina 29)
- [24] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340. (Citato a pagina 30)

Questa tesi ha contribuito a piantare un albero in Kenya tramite il progetto Treedom.

<https://www.treedom.net/it/user/sesalab/event/sesa-random-forest>