Code Generation of Smart Contracts with LLMs: A Case Study on Hyperledger Fabric

Luca Olivieri
Ca' Foscari University of Venice
Venice, Italy
luca.olivieri@unive.it
ORCID: 0000-0001-8074-8980

David Beste
CISPA Helmholtz Center for Information Security
david.beste@cispa.de
Saarbrücken, Germany
ORCID: 0009-0002-0597-7788

Luca Negrini
Ca' Foscari University of Venice
Venice, Italy
luca.negrini@unive.it
ORCID: 0000-0001-9930-8854

Lea Schönherr
CISPA Helmholtz Center for Information Security
lea.schoenherr@cispa.de
Saarbrücken, Germany
ORCID: 0000-0003-3779-7781

Antonio Emanuele Cinà
University of Genoa
Genoa, Italy
antonio.cina@unige.it
ORCID: 0000-0003-3807-6417

Pietro Ferrara
Ca' Foscari University of Venice
Venice, Italy
pietro.ferrara@unive.it
ORCID: 0000-0002-4678-933X

***Abstract*—Hyperledger Fabric (HF) is currently the one that made blockchain and smart contracts accessible to industries, providing highly customizable solutions for many enterprise use cases. Despite this, programmers are often discouraged from implementing smart contracts due to the high learning curve and security risks of naive smart contract implementations. At the same time, the advent of Large Language Models (LLMs) for code generation led to new possible scenarios such as creating new smart contract applications starting from natural language, allowing to reduce costs and development times. This paper investigates the maturity of LLMs for the code generation of HF smart contracts. In particular, we (i) generate smart contracts written in Go for HF starting from natural language descriptions, (ii) select state-of-the-art static analyzers of Go program, and (iii) perform a quality and security assessment of the generated smart contracts. Our empirical results show current LLMs do not produce high-quality smart contracts, and a relevant effort to debug and patch contracts containing bugs and possible vulnerabilities.**

***Index Terms*—LLM, Code Generation, Blockchain, Smart Contract, Chaincode, Static Analysis, Program Verification**

I. Introduction

Over the last decade, the blockchain and smart contracts attracted the attention from industries, governments, and academia [1]–[5]. Thanks to these technologies, it is possible to collect information in a tamper-proof way and without the need for third-party intermediaries allowing to reduce costs and make safer operations (e.g., asset transfer [6], agreement stipulation [7], ...). However, while having an anti-tamper system such as the blockchain is a great security feature, it can also make it hard to patch vulnerabilities, unexpected behaviors, and wrong data after the deployment and execution of buggy smart contracts, leading in the worst cases to huge economic losses [8]. For these reasons, smart contract implementations require rigorous development to ensure a bug-free deployment in the blockchain.

Fueled by substantial success in natural language applications, Large Language Models (LLMs) have demonstrated unparalleled proficiency in diverse domains [9]–[11]. One notable application is their integration into code assistants, where they are extensively trained on expansive open-source code repositories to generate functionally correct programs [12], [13]. These models generate functionally correct programs based on user-provided prompts [14], [15]. This innovative utilization forms the foundation of numerous commercial code completion engines, reshaping the programming landscape and enhancing developer productivity [16]. For example, the Codex model [15] is the driving force behind GitHub Copilot [14], a widely utilized tool in the coding community. Various studies have affirmed the positive impact of these language models on programming productivity [16]–[18]. As LLMs assert their prowess in supporting code development, a natural extension of their capabilities leads us to explore their potential in smart contract development. However, a critical caveat emerges, i.e., their lack of awareness regarding security concerns. Researchers have indeed recently demonstrated that LLMs can generate code that may compromise security standards [19]. In the context of smart contracts, code quality and security are paramount because, depending on the type of blockchain, they can be immutable or upgradable but with non-trivial consequences. For these reasons, it is necessary to guarantee code that is as optimized and safe as possible, and it is recommended that software verification be applied by design. Moreover, this last factor becomes more problematic because of the lack of standard development architectures and best practices [20]–[22].

This raises concerns about the reliability of LLM-generated code, especially in enterprise blockchain frameworks such as Hyperledger Fabric, where high standards of security, data integrity, and business requirements must be guaranteed.

We explore the intricate complexities of smart contract development, aiming to ascertain whether LLMs can meet the stringent safety and reliability requirements in the blockchain technology domain. Through rigorous testing, including all publicly available static analyzers for Hyperledger Fabric and 6 distinct LLMs, we seek to contribute valuable insights into the potential and limitations of LLMs in creating smart contracts, ultimately addressing the imperative need for enhanced awareness and precision in their generative processes.

Methodology: The main goal of our work is to investigate the smart contracts produced by LLMs, focusing on the Hyperledger Fabric framework, and to assess their quality and security. To do so, we will apply static analyzers representing the state-of-the-art in this field for Hyperledger Fabric smart contracts. Our experimental evaluation will be focused on the following three research questions:

RQ1 Are smart contracts generated by LLMs reliable and secure?

RQ2 Do code LLMs produce better (that is, more reliable and secure) smart contracts? Do commercial LLMs produce better smart contracts?

RQ3 How to assess the quality and security of the generated smart contracts, and what kinds of tools are most effective?

To answer these questions, we selected both open-access and commercial LLMs (see Section III). From the static analysis perspective, we applied both utility and verification tools to the smart contracts generated by the LLMs (see Section IV).

Paper structure: Section II introduces background notions related to the paper. Section III deals with generating smart contracts through large language models. Section IV provides an overview of verification tools that we involved in the assessment and security issues investigated. Section V describes the experimental results of the assessment performed on the smart contracts generated by large language models. Section VI discusses related work. Section VII concludes the paper.

II. Technical Background

This section provides a high-level and comprehensive overview of the key concepts essential for understanding the contents and terminologies used throughout this paper.

A. Blockchain and Smart Contracts

Blockchain technology is a distributed and decentralized computing architecture in a network where peers approve or reject transactions towards that architecture through a consensus mechanism. The architecture's primary goal is to record transaction information and data in an immutable manner and put them into a ledger structured as a chain of blocks.

Typically, the blockchain architecture also supports a framework to handle *smart contracts*, i.e., computer programs that can be stored and executed within the blockchain, allowing, for instance, to automate tasks, to make custom business logic for the users, to perform agreement between the parties, etc.

1) Hyperledger Fabric: Hyperledger Fabric (HF) [23], [24] is currently the most popular framework for the development of enterprise-grade *permissioned* blockchain solutions. Compared to other mainstream blockchains such as Bitcoin [6] and Ethereum [7], it has become popular due to its flexibility, scalability, and network settings, which allow solutions to be tailored to specific industry needs. As reported by IBM company [24], many organizations contributed to the HF project, hosted by the Linux Foundation and adopted by the mainstream cloud providers (e.g. IBM, AWS, Azure) [25].

In HF, smart contracts are also known as *chaincodes* and can be written in several Turing-complete general-purpose programming languages, where the most adopted one is Go. In this paper, we use the more generic term, i.e., *smart contract*, but feel free to interpret it as *chaincode* if that concept is more familiar. The contracts are deployed and run on different *channels*, i.e., sub-networks with their own blockchain ledgers and policies. A ledger consisting of two components: the world state (mutable), a key-value database which stores the current state of contracts, and the blockchain-based transaction log (immutable), which maintains the history of all transactions.

B. Software Static Analysis and Verification

The assessment of the quality and security of software has been a relevant topic both in the scientific community and in industry during the last few decades, and several approaches to detecting bugs and vulnerabilities have been developed and successfully applied to real-world software [26]. In this work, we focus on static analysis [27], i.e., to inspect the code without running it.

Since static code analysis does not need to build up concrete states to execute a program, it usually achieves better coverages compared to dynamic analysis [27], as well as it does not require complex architectures and test environments to perform the analyses. Furthermore, dynamic testing of smart contract code written by an LLM is difficult due to the lack of environment information, such as blockchain state, external contract interactions, and execution context, which can lead to incomplete or misleading test results. In addition, LLM-generated code may contain types of problems that might prevent its execution and dynamic testing, since the programs returned are usually partial and do not comprehend all the code needed for their execution.

Smart contracts have been a focus of several works in this field [28]–[30]. In addition, more in-depth studies have also been performed for Hyperledger Fabric [31]–[33]. According to [34], smart contract verification is challenging due to blockchain properties (anti-tampering, decentralization, and distribution) and the use of general-purpose languages further increases the issues. Therefore, it is not certain that the code can be fixed as easily and quickly as with traditional software and requires to ensure security as early as possible in the software development.

C. Large Language Models

Large Language Models (LLMs) are machine learning models that leverage the capabilities of transformer architectures [35] for language processing tasks. LLMs are increasingly becoming an essential component in the development of robust tools that leverage foundation models, either directly trained in code or natural language, for downstream tasks such as scammers detections [5], [36], [37], code generation [15], code repair [38], [39], and reverse deobfuscation [11]. Typically trained on extensive datasets of natural language or code [9], can acquire fundamental and general language understanding and generation skills. Additionally, they develop in-context learning capabilities, enhancing their ability to adapt to various tasks and contexts. The most common objective to pre-train LLMs, such as GPT3 [9] and PaLM [40], is the language modeling task, also called the autoregressive language pre-training method. The autoregressive models, the focus of this work, predict the next token—textual units, such as letters, words, or sub-words—from a sequence of input tokens. Specifically, autoregressive foundation LLMs have been proven to be particularly advantageous in scenarios requiring few-shot or zero-shot text generation capabilities [41]. For instance, both GPT-3 [9] and PanGu-α [42] have showcased robust performance in zero-shot and few-shot learning scenarios [43].

LLMs for Code Assistant.: In this work, we utilize different code language models, which are trained to generate functional code from a prompt describing the task of a function or code unit. Specifically, recent works have introduced several LLMs designed for code modeling, including DeepSeek [44], CodeGen [13], CodeLLama [12], and others [14], [15], [40]. The models can generate appropriate code in different programming languages and complete the code according to the context description provided. This is accomplished by using a text prompt and a function header as system input, i.e. input to the model.

III. Generation of Contracts with LLMs

To the best of our knowledge, no publicly available dataset exists that provides programming language-agnostic smart contract descriptions in natural language. To address this gap, we first constructed a dataset from scratch by combining real-world examples with synthetically generated descriptions produced via GPT-4. The dataset consists of 1,345 natural language descriptions and names of smart contracts spanning more than 50 application domains (e.g., finance, healthcare, gaming, automotive, supply chain, sustainability, etc.). Of these, 73 descriptions were collected from real-world sources, while 1,272 were generated synthetically using the following GPT-4 prompt: In particular, the user prompt designed used for the specific synthetic smart contract description is the following:

```
Write the descriptions in natural language of 100 smart
    contracts for [APP_DOMAIN]
```

where the variable [APP_DOMAIN] denotes the name of an application domain. Finally, the generation with LLMs computed 8063 smart contracts.

The natural language descriptions from this dataset were then used as input to various LLMs (both open-access and commercial) to generate smart contracts in Go for the Hyperledger Fabric framework [47]. In total, 8,063 smart contracts were generated. The LLMs evaluated are summarized in Table I. All models were configured with temperature $T = 0$ to minimize syntax errors, which are more common at higher temperatures, particularly in smaller models. The prompt design was adapted to each model to ensure deterministic and syntax-compliant outputs. Coding-specialized models (DeepSeek-Coder, CodeLlama) required no special system prompts; a simple prefix was sufficient to consistently generate valid Go code for Hyperledger Fabric. Claude 3.5, being a pure code model, required iterative refinement to enforce the desired format. In contrast, GPT-4 required a custom system prompt due to the lack of strict output control at the time:

TABLE I: Information of selected LLMs

Name	Open Access	Provider	# Parameters (Billions)
deepseek-coder-6.7-instruct [44]	✓	DeepSeek	6.7
deepseek-coder-33-instruct [44]	✓	DeepSeek	33
CodeLlama-7b-Instruct-hf [12]	✓	Meta	7
CodeLlama-34b-Instruct-hf [12]	✓	Meta	34
Claude 3.5 [45]	✗	Anthropic	Unknown
GPT-4 [46]	✗	OpenAI	Unknown

```
System Prompt (GPT-4):
"You are a smart contract programmer and
    always start your response with
\"Sure, here is the code in Go for
    Hyperledger Fabric framework: ```go\""
```

IV. Setup of the Assessment

In this section, we discuss how we chose the smart contracts, tools, and issues that will be the target of our assessment. Our focus is verifying code quality and security, which pertain to non-functional requirements, rather than the specific behaviors related to business or application logic, which are functional requirements. Furthermore, it is orthogonal to our study to evaluate the semantics of natural language description compared to the semantics of the generated code. In fact, it is not possible to apply automatic tools to verify their correctness since the dataset definitions are not written in a formal way. Moreover, there are theoretical limits to what can be determined automatically, such as non-trivial semantic properties, as outlined by the Rice's theorem.

A. Inspection of generated smart contracts

Figure 1 summarizes the findings of this section.

First, we analyzed the contracts to understand if they are syntactically correct and, therefore, potentially compilable. For this operation, we used *gofmt* [48], a tool that performs a static analysis of the code and returns an error if the analyzed program does not satisfy the grammar of the Go language. In this way, we distinguished well-formed smart contracts from ill-formed ones. Then, we manually investigated the code of the general smart contracts to check if they contained some application logic. In particular, during the data preparation process, two researchers, with experience in smart contract development and LLMs for code generation, independently reviewed a subset of the generated code to identify contracts with actual application logic. They looked for elements such as state manipulation, conditional logic, and domain-relevant operations. Disagreements were resolved through discussion

TABLE II: Average lines distribution per generated contract

LLM	Blank Lines	Comment Lines	Physical LoCs
CodeLlama	60.34	53.66	244.43
CodeLlama34	49.55	37.06	216.19
DeepSeek-Coder	18.88	0.21	69.80
DeepSeek-Coder33	19.04	4.11	87.84
Claude3.5	29.40	8.99	130.87
GPT-4	21.77	0.22	81.23

to ensure consistency. In this way, we filtered out trivial or ill-formed outputs before applying static analysis, not to assess correctness in a formal sense.

We also identified duplicated contracts which comprised of less than 3% of the generated data. Although they were generated by different sentences written in natural language, the generated contracts have the same MD5 hashes. In checking them, we believe the main reason is that the model is confused by similar contract descriptions. Another aspect we observed is that some samples exhibit truncated code. This issue can generally be attributed to two root causes: either the specified token size was insufficient for the model to generate a complete, well-formed contract, or the model became stuck in a repetitive loop, a known phenomenon associated with maximization-based decoding strategies [49]. One potential mitigation is to employ a different sampling strategy, which can reduce the occurrence of this issue. However, exploring alternative decoding strategies is beyond the scope of this work. For samples where the truncation was due to the first issue, increasing the maximum token size, if supported by the model, resolves the problem.

Finally, we collected statistics on the lines of code (LoCs) using the tool *cloc* [50]. Table II provides the average lines of code per file. Note that although the generated code for each smart contract is small in size, it accurately reflects real smart contracts. For instance, many deployed contracts in the Ethereum main network consist of fewer than 200 instructions [51]. In terms of LoCs, this is very close to the averages of our generated contracts, as there are typically multiple instructions per line.

B. Verification Tools

Regarding smart contracts, there are several studies on Hyperledger Fabric issues [31]–[33], but currently, only a few analyzers are capable of detecting them. Moreover, as reported in Section II-B, we are interested in static analyzers that automatically examine the code without executing the program. For instance, Chaincode Analyzer [52], by Fujitsu from Hyperledger Labs, and Reviveˆ CC [53], an extension of the Revive analyzer [54] for chaincodes. Both tools represent the program using abstract syntax trees (ASTs), applying static intra-procedural analyses on them and using a syntactical approach to detect different issues (such as non-determinism, read after write, . . .). Lv et al. [31] and Yamashita et al. [32] propose similar tools inspired by Chaincode Scanner and Reviveˆ CC, but they only perform syntactic checks although in a more accurate way.

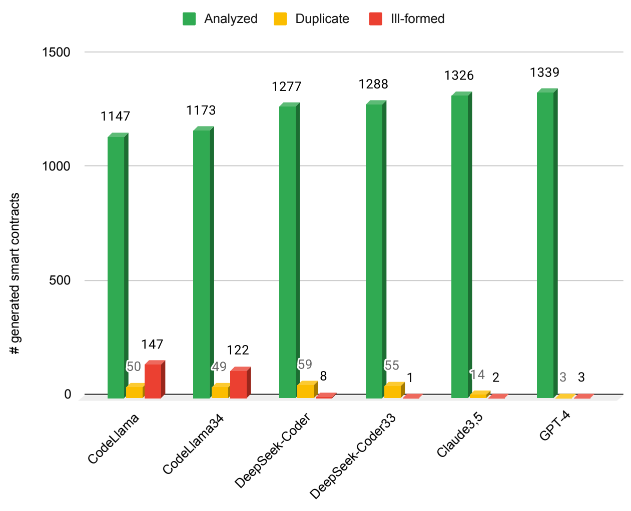

Fig. 1: Distribution of the generated contracts analyzed with *gofmt* and *cloc*.

Instead, a semantic-based approach is followed by ZEUS [55] and GoLiSA [56], [57]. Both tools propose semantics-based static analysis approaches based on abstract interpretation theory [58], [59]. Abstract interpretation provides a systematic way to approximate program semantics with an abstract version of them and to reason about some program properties, achieving mathematical guarantees during the analysis, such as ensuring the presence or absence of certain code properties, bugs, and vulnerabilities.

Tools providing a primarily syntactic analysis (Chaincode Analyzer, Reviveˆ CC, Lv et al., and Yamashita et al.) perform quickly and yield straightforward results. Since these tools can only analyze how a program is written, they are typically limited to either (i) searching for known patterns that are indicative of security issues or (ii) prohibiting the usage of entire APIs and language constructs to ensure the absence of exploits and errors.

However, an assessment of the code quality and security cannot only include syntatic analyses because searching for patterns is inherently *unsound*: as with any bug, a security issue may appear in various forms, and attempting to enumerate them all within such a tool is impractical. Consequently, aforementioned tools may overlook certain contract issues, making insecure contracts mislabeled as secure. On the other hand, prohibiting the use of certain APIs and language constructs severely restricts a developer's options. Developers are left with the choice of either following the tool's decision not to use it or dealing with its numerous warnings.

For these reasons, we also considered a tool with semantic-based analyses (such as ZEUS and GoLiSA), which can provide guarantees regarding their findings. However, computing a program's semantics is generally undecidable. Since the semantics cannot be precisely computed, such tools operate on an over-approximation of it to maintain computability, which

TABLE III: Issues covered by the tools

Issue	Revive^CC	GoLiSA
Non-determinism	✓	✓
Read-Write Issues	✓	✓
Phantom Reads	✗	✓
Unhandled Errors	✗	✓
Untrusted Cross-Contract Invocations	✗	✓
Cross-Channel Issues	✗	✓
Numerical Overflows	✗	✓
Divisions By Zero	✓	✓

can result in false positives due to the inherent limitations of the approximation itself.

In this paper, we use both categories of tools to increase the likelihood of detecting security issues by applying different checks. In particular, we choose Revive^CC, and GoLiSA as linter and semantic-based tool, respectively. The reason is that they were the only publicly available tools among the aforementioned ones together with Chaicode Analyzer, which however we excluded because it failed in more than 60% of analyses due to mainly parsing errors.

Table III presents, for each tool, whether they cover the security issues considered in this work or not, while Table IV provides a summary description of them.

V. Experimental Results

In this section, we report our findings when analyzing the smart contracts generated with LLMs (Section III) with Revive^CC, and GoLiSA, aiming at discovering the issues presented in Table IV. All analyses have been performed on a machine equipped with an AMD Ryzen 9 3950X 16-Core at 3.50 GHz, 128 GB of RAM, 20 TB SSD, running Ubuntu 22.04.3 LTS, Open JDK version 21. During the analysis, 8 GBs of RAM were allocated to the Java Virtual Machine. As targets of the analyses, we used the generated well-formed smart contracts without duplicates, i.e., 7550 smart contracts.

We perform a quantitative evaluation to answer the research questions listed in Section I, and a qualitative evaluation presenting several examples of the issues found by the tools on the generated smart contracts.

A. Benchmark Coverage

Table V reports for each verification tool, the number of contracts analyzed successfully, the number of contracts partially analyzed (the analysis was successful for some checks, while not for others), the number of contracts on which the analysis failed (all checks led to analysis failures), the number of contracts affected by at least one vulnerability, and the number of warnings generated by the analysis.

Before discussing the table, it is worth noting that (i) some contracts generated by CodeLlama miss to import packages of methods that are used in the code, and (ii) the majority of contracts generated by CodeLlama and few of the contracts generated by DeepSeek-Coder and DeepSeek-Coder33 contain typing errors (as an example, consider Figure 2 where a string is first compared with a numerical quantity and then used inside a subtraction). Thus, it is expected that the tools

```go
user1Bytes, _ := APIstub.GetState(user1)

user1Balance := string(user1Bytes)

if user1Balance < amount {
    // ...
}

user1NewBalance := user1Balance - amount
```

Fig. 2: Typing errors in *deepseek-coder/1115.go*.

highlight these situations by not issuing warnings for the former case and by either failing the analysis or issuing warnings for the latter case.

Revive^CC and GoLiSA successfully process and analyze the majority of the contracts under analysis: Revive^CC fails on 4 contracts only (due to runtime-exceptions), while GoLiSA analyzes successfully more than 93% of smart contracts (7071 out of 7550), partially around 6% (473 out of 7550), and fails to analyze less than 1% (6 out of 7550). In GoLiSA, smart contracts that are partially analyzed or fail to perform all analyses are primarily due to increased analysis complexity, timeouts that stop execution after an excessive amount of time (set to 10 minutes), and other tool exceptions. In terms of warnings generated, GoLiSA issues sensibly more warnings than Revive^CC: this is not only due to Revive^CC missing checks for four and three of the targeted security issues, respectively, but also due to the deeper analysis performed by GoLiSA's semantic engine.

B. Quantitative evaluation

In this section, we classified the warnings coming from the analysis results. To this end, as usual for static analysis benchmarks, we set the ground truth performing an in-depth manual investigation of the code of the generated smart contracts and the program points highlighted by analysis reports to determine what the real issues were. Specifically, they were manually inspected by a software security expert, specialized in blockchain and smart contract software, who classified them as true or false positives. The classification was based on the formal definition of the various properties detected by the analyzers. Table VI reports the number of true positives (that is, warnings reporting real vulnerabilities that can happen during the contract's execution)) for each verification tool on contracts generated by each LLM by Revive^CC, and GoLiSA, respectively.

Column ND reports warnings about non-determinism, column RW reports warnings about read-write set issues, column UE shows warnings related to unhandled errors, column UCCI displays warnings about untrusted cross-contract invocations, column OVF reports warnings about overflows and underflows, and column DIV reports warnings on divisions by zero.

Cells containing - highlight the missing coverage of the tool for that issue. The number of false positives (that is, warnings reporting not real vulnerabilities) of each analyzer

TABLE IV: Description of HF issues

Issue	Description
Non-determinism (ND)	Non-deterministic values (i.e., a value that varies among different peers) could lead to critical consequences when they are candidates for writing into the world state, because the consensus mechanism might reject the transaction associated with that smart contract executions [57], [60].
Read-Write Issues (RW)	*Read-your-writes* semantics [61] are not supported in HF. If multiple writes occur for a given key, only the last one executed will be committed to the ledger state. If the value for a given key is read, it is retrieved from the committed state, even if a write was previously executed to update the value of that key [62].
Phantom Reads (PR)	Transactions may query the world ledger state with specific APIs, but during execution, another transaction modifies the data set, leading to performance bottlenecks and unexpected results during validation [63].
Unhandled Errors (UE)	Errors that are not explicitly handled may expose smart contracts in vulnerable or unexpected states [64].
Untrusted Cross-Contract Invocations (UCCI)	Cross-contract invocations must be carefully managed to prevent arbitrary code executions and unexpected behaviors [65]–[67].
Cross-Channel Issues (CCHI)	Cross-contract invocations within different channels may lead to a lack of transparency and missing commitments in the world ledger state [31], [32]
Numerical Overflows (OVF)	As computers have finite memory, there is a limit to the numbers that can be represented. When such limit is breached a numerical *overflow* (or *underflow*) occurs [68]. In general, overflows and underflows may have critical consequences in the blockchain context. For instance, an attacker can exploit it by repeatedly invoking a smart contract function with a numerical overflow bug that increases a value, to drain more money than it should [69], [70].
Division by Zero (DIV)	If a smart contract does not handle division by zero properly, it can cause runtime errors and transaction failures.

TABLE V: Raw analyses results for verification tools

	Revive^CC				GoLiSA				
Model	**# Fully Analyzed**	**# Failures**	**# Affected**	**# Warnings**	**# Fully Analyzed**	**# Partially Analyzed**	**# Failures**	**# Affected**	**# Warnings**
CodeLlama	1147	0	24	28	1009	137	1	125	326
CodeLlama34	1171	2	28	109	1011	160	2	155	403
DeepSeek-Coder	1278	0	7	9	1236	41	0	355	549
DeepSeek-Coder33	1288	0	3	3	1226	62	0	550	912
Claude3.5	1324	2	605	620	1277	46	3	764	1738
GPT-4	1339	0	76	82	1312	27	0	323	547

been omitted for clarity, but their rate (percentage w.r.t. the total number of warnings issued by the tool) will be reported in the following. Revive^CC raises true positives (687 alerts) about non-determinism only, and none about the other issues they support. The tool has also the ∼19% of false positive rate. Instead, GoLiSA raises a total of 4012 true positives, with a false positives rate of ∼10%. While the number of true positives might seem excessively high w.r.t. the Revive^CC, it is worth noting that the majority of them are issued on unhandled errors and overflows/underflows, that is, on issues that Revive^CC does not cover. The difference between the syntactic and semantic approach is evident: while incurring in a higher computational cost, semantic analyzers like GoLiSA are able to achieve a deeper understanding of the behavior of each program and can thus report more significant warnings to the users. In fact, thanks to GoLiSA's findings, we can rule almost 29% (2157 out of 7550) of the contracts generated by the LLMs considered in this work as insecure, since they contain at least one vulnerability. Note that this number does not take into account invalid contracts (e.g., ones without code or with typing errors, as discussed in Section V-A).

a) RQ1: Figure 3 plots the percentage of smart contracts that are affected by different types of issues identified by static analysis means . This plot underlines that(i) some issues does not appear (i.e. PR), (ii) some issues (UCCI, CCHI, and DIV in particular) appear only on very few smart contracts with less than 1%, (iii) other issues (RW, ND and OVF) appear on a non-

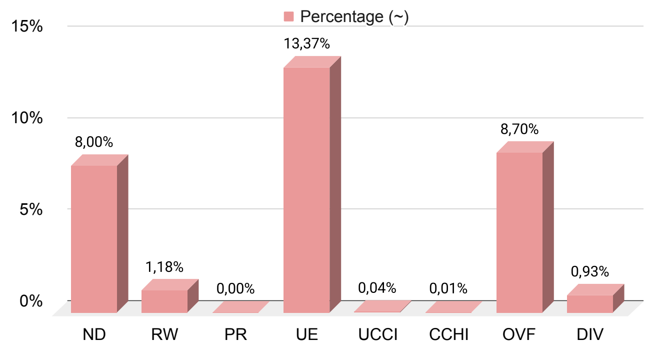

Fig. 3: Percentage of contracts affected by each issue.

negligible subset (between 1% and 9%), and (iv) unhandled read-write errors (UE) affects almost a tenth of the contracts generated. This shows that generated smart contracts are not secure and reliable since they need a deep revision to catch and fix these issues. Still, such a revision can be focused on some specific properties and partially automated using existing static analysis tools.

b) RQ2: Figure 4 plots the number of true positive warnings produced by all the analyzers on the smart contracts generated by different LLMs. Our experiments applied 6 different LLMs, four open-source ones targeting the code generation (CodeLama, CodeLama34, DeepSeek-Coder, and

TABLE VI: True positives per issue

Model	Revive^CC ND	Revive^CC RW	Revive^CC DIV	GoLiSA ND	GoLiSA RW	GoLiSA PR	GoLiSA UE	GoLiSA UCCI	GoLiSA CCHI	GoLiSA OVF	GoLiSA DIV
CodeLlama	3	0	0	6	7	0	34	0	0	108	7
CodeLlama34	89	0	0	17	57	0	70	0	0	87	6
DeepSeek-Coder	6	0	0	10	63	0	334	2	0	112	2
DeepSeek-Coder33	3	0	0	1	6	0	846	2	2	41	0
Claude3.5	533	0	0	880	0	0	3	0	0	708	70
GPT-4	52	0	0	59	0	0	320	0	0	141	11

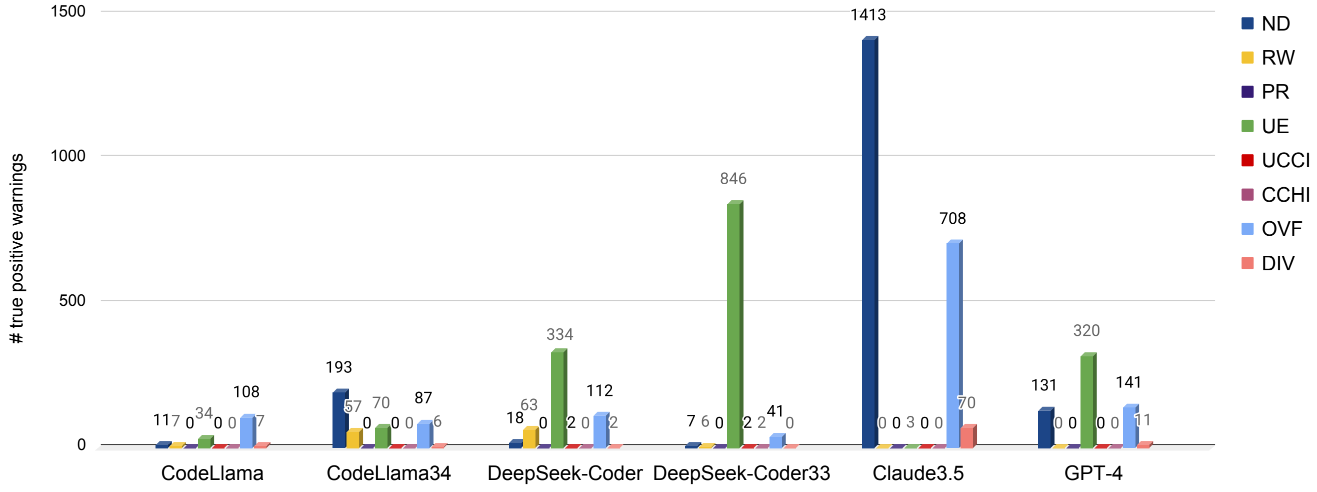

Fig. 4: True positive warnings in the contracts per LLM.

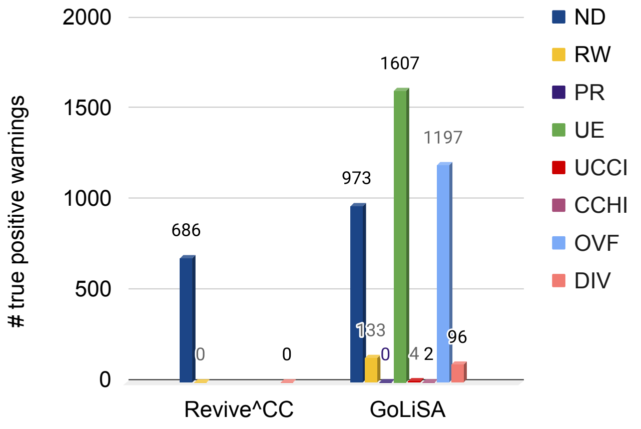

Fig. 5: True positive warnings produced by the analyzers

DeepSeek-Coder33), and two generic and commercial (Claude 3.5, GPT-4) automatically. The plot underlines that the security of the generated smart contracts is comparable since the number of true positives produced on the code generated by different LLMs is comparable. While GPT-4 performs slightly worse than the others, it shows that even for generic LLMs, it is possible to produce smart contracts as secure as the ones produced by code-specific LLMs. In addition, it is not surprising that overall, DeepSeek-Coder33 (using 33 billion features) performs quite better than DeepSeek-Coder (using only less than 7 billion parameters). However, it contains many more issues with unhandled read-write errors, and it is worth further investigation to understand why this happens. Our assessment does not show that a specific type of LLM (commercial or code-specific) produces better smart contracts.

c) RQ3: Figure 5 plots the number of true positive warnings each analyzer produces for each type of analysis. The plot makes evident that (i) the amount of positives produced about numerical overflows and unhandled read-write errors exceeds by far the other warnings, (ii) only GoLiSA (that is, a semantic analyzer) detects such issues, and (iii) the amount of reports about other properties is comparable among different analyzers. Note that GoLiSA is the only semantic analyzer. On the one hand, numerical overflows and unhandled read-write errors require deep (semantic) analyses to achieve an acceptable precision; therefore, only GoLiSA detects them. On the other hand, there is no relevant difference between syntactic and semantic analyzers for other properties. For instance, non-determinism can be easily detected on small smart contracts produced by LLMs by checking if a non-deterministic library is called. Note that semantic analysis comes at a cost: it requires more computational resources and time, and it might support only a restricted set of smart contracts (e.g., even if a program does not type check, a syntactic analyzer usually can

```
1  var machine = Machine{ID: args[0],
       Condition: args[1], LastMaintenance:
       time.Now()}
2
3  machineAsBytes, _ := json.Marshal(machine)
4  APIstub.PutState(args[0], machineAsBytes)
```

Fig. 6: Issue of non-determinism in *gpt-4/1013.go*.

process it, while a semantic one cannot). Overall, we conclude that for simple analyses, existing analyzers assess the security of smart contracts similarly, while for more complex properties (needed to perform a deep and reliable assessment), semantic analyzers, such as GoLiSA, are needed.

C. Qualitative evaluation

In the following, we show how examples of the security issues described in Table IV were discovered inside the target benchmark.

1) Non-Determinism: Figure 6 reports a non-deterministic contract generated by GPT-4. At line 1, a machine is created settings several values, and in particular `LastMaintenance` using the method `time.Now()`. Then, the machine object is converted into bytes at line 3, and its storage in the blockchain is proposed using `PutState` at line 4. In this case, the source of non-determinism is to the method `time.Now()`, as it depends on the local machine clock, and it may differ depending on the machine on which it is executed. The reason is that, in the consensus mechanism, the involved peers (who have potentially different machines) may not reach a consensus due to too many different time values, leading to the failure of the transaction [57].

In this case, Revive^CC detects syntactically the `"time"` import at the beginning of the program thought pattern matching. Instead, GoLiSA performs an information flow analysis [57] starting from `time.Now()`, then `machine`, until `APIstub.PutState(args[0], machineAsBytes)` triggering an alarm on it because a non-deterministic value has been propagated at blockchain write operation thought `machineAsBytes`.

2) Issues of Read-Write Set Semantics: Figure 7 contains a smart contract generated by DeepSeek-Coder. At line 9, the method `transferToken` performs a transfer of an amount of tokens from one user to another. The user information are collected in the variables `from` and `to` at lines 11-12, which are coming from the transaction input retrieved by `GetFunctionAndParameters()` at line 2. Therefore, being an arbitrary user input, they can be assigned any value. The token transfer happens at lines 13-16, where `GetState(from)` yields the current token value of `from` user from the ledger, `PutState(from, []byte(value - args[2]))` removes the value `args[2]` from `from` user and writes the results on the ledger, `GetState(to)` yields the current token value of `to` user from the ledger, and `PutState(to, []byte(value + args[2]))` adds the value to `to` user.

```
1  func (s *SmartContract) Invoke(APIstub
       shim.ChaincodeStubInterface) sc.Response
       {
2    function, args :=
         APIstub.GetFunctionAndParameters()
3    // ...
4    } else if function == "transferToken" {
5      return s.transferToken(APIstub, args)
6    // ...
7  }
8  // ...
9  func (s *SmartContract)
       transferToken(APIstub
       shim.ChaincodeStubInterface, args
       []string) sc.Response {
10   // ...
11   from := args[0]
12   to := args[1]
13   value, _ := APIstub.GetState(from)
14   APIstub.PutState(from, []byte(value -
         args[2]))
15   value, _ = APIstub.GetState(to)
16   APIstub.PutState(to, []byte(value +
         args[2]))
17   // ...
18 }
```

Fig. 7: Read-Write set issues in *deepseek-coder/1245.go*.

In this contract, there are read-after-write and over-write issues that can be exploited by an attacker to increase its amount of value. Since `from` and `to` can have any value, nothing prevents an attacker from creating an input where the value of `from` is the same as `to` (transaction to oneself). In this scenario, suppose to have an initial state of ledger `{"attacker": 10}`, where the attacker has 10 units of value. If the attacker sends the transaction input `args = {"attacker", "attacker", 5}`, we have that `from = "attacker"` and `to = "attacker"` at lines at lines 11-12, while `value = 10` at line 13. Then, `PutState("attacker", []byte(10 - 5))` happens at line 14, but `value = 10` at line 15 (read-after-write issue) as it is still read from the pre-execution state. This is followed by `PutState("attacker", []byte(10 + 5))` at line 16. Moreover, `PutState` instructions at lines 14 and 16 have the same user `"attacker"`: the only value written in the ledger is `10+5` (over-write issue). Therefore, at the end of the computation of the method `transferToken`, no subtraction is done for the attacker, and the 5 units of value are added instead of keeping the attacker's value unchanged.

Regarding the detection of read-write set issues, Revive^CC detects only read-after-write issues. Their work matches method signatures. Then, for each read-write tuple, they check the actual parameters in each key position and perform a syntactic check on the token syntax of actual parameters. However, this approach leads to some limitations. For instance, two variables used as keys with an equal stored value but named differently will be considered different by the analyses, leading not to detection of the issue as in the case of Figure 7.

```
1  func (s *SmartContract) transferAsset(stub
       shim.ChaincodeStubInterface, args
       []string) peer.Response {
2      // ...
3    assetAsBytes, _ = json.Marshal(asset)
4    stub.PutState(args[0], assetAsBytes)
5
6    return shim.Success(nil)
7  }
```

Fig. 8: Unhandled error in *deepseek-code33/1197.go*.

Instead, GoLiSA detects both read-after-write and over-write issues. In particular, GoLiSA over-approximates the possible string values of keys exploiting a string value analysis. Then, after the computation, the analyzer inspects the position of read and write operations traversing the control-flow graphs of the program and, in case, triggers the alarms.

3) Phantom Reads: No issues related to phantom reads were found in the dataset.

4) Unhandled Read-Write Errors: The smart contract in Figure 8, generated by DeepSeek-Coder33, ignores possible errors during its execution. The goal of `transferAsset` is to transfer an asset from one owner to another. At line 4, the writing is carried out on the ledger by the method `PutState`. However, any return errors of the `PutState` are not collected in a variable (e.g. `err := stub.PutState(args[0], assetAsBytes)`) and after checked. Furthermore, the `transferAsset` returns always `shim.Success` (line 6), i.e. that the transaction was successful. Hence, if the `PutState` fails and it does not write the value in the ledger, the transaction would be mistakenly considered successful. This leads to a serious logical error because a user might think that the asset was transferred correctly when, in fact, it was not transferred due to an error.

Regarding verification tools, Revive^CC does not provide checks for the detection of unhandled errors. Instead, GoLiSA performs a check to detect blank identifiers[1] and unassigned error values of blockchain read and write operations.

5) Untrusted Cross-Contract Invocations: The snippet in Figure 9 shows a smart contract generated by DeepSeek-Coder. At line 2, it retrieves the input of a transaction request with the function `GetFunctionAndParameters`. At line 4, it calls the method `swapAssets`, and inside its body the input is assigned to the variables `chaincodeName`, `chaincodeFunc`, `assetID`, `newOwner` (lines 8-11). Then, the values of variables are used to perform `InvokeChaincode`, a cross-contract invocation at line 13. In this case, the input provided by `GetFunctionAndParameters` is considered untrusted because any user could create a transaction passing an arbitrary input to it. Moreover, input checks or sanitizers are not present in the contract: an attacker could create an input providing a malicious value for `chaincodeName` (i.e.

```
1  // ...
2    function, args :=
         APIstub.GetFunctionAndParameters()
3    if function == "swapAssets" {
4      return s.swapAssets(APIstub, args)
5  // ...
6  func (s *SmartContract) swapAssets(APIstub
       shim.ChaincodeStubInterface, args
       []string) sc.Response {
7    // ...
8    chaincodeName := args[0]
9    chaincodeFunc := args[1]
10   assetID := args[2]
11   newOwner := args[3]
12
13   response :=
         APIstub.InvokeChaincode(chaincodeName,
         [][]byte{[]byte(chaincodeFunc),
         []byte(assetID), []byte(newOwner)}, "")
14   // ...
15 }
```

Fig. 9: UCCI in *deepseek-coder/832.go*.

```
1  func (s *SmartContract)
       transferAsset(APIstub
       shim.ChaincodeStubInterface, args
       []string) sc.Response {
2    // ...
3    crossChainBridgeAsBytes, _ :=
         APIstub.GetState(args[0])
4    crossChainBridge := CrossChainBridge{}
5
6    err :=
         json.Unmarshal(crossChainBridgeAsBytes,
         &crossChainBridge)
7    // ...
8    response := APIstub.InvokeChaincode(
9      crossChainBridge.ChaincodeName,
           chaincodeArgs,
           crossChainBridge.ChannelName)
10     // ...
11 }
```

Fig. 10: Cross-channel issues in *deepseek-code33/653.go*.

the smart contract where the assets are transferred), diverting the asset to another owner and effectively stealing the asset.

Regarding verification tools, Revive^CC does not provide checks for the detection of UCCIs. Instead, GoLiSA tracks untrusted information in the program, exploiting also, in this case, an information-flow approach.

6) Cross-Channel Issues: The snippet in Figure 10 shows a smart contract generated by DeepSeek-Coder33, containing potential cross-channel issues. The goal of `transferAsset` is to transfer an asset from one owner to another. At line 3, `GetState(args[0])` reads the value of an arbitrary key coming from an transaction input and it is used to populate the object `CrossChainBridge` at line 6. Then, the value of `CrossChainBridge.ChannelName`

[1]https://go.dev/ref/spec#Blank_identifier

```
1  func (s *SmartContract)
     distributeDividends(APIstub
     shim.ChaincodeStubInterface, args
     []string) sc.Response {
2    //...
3    totalShares := 0
4    // ...
5    totalShares += shareholder.Shares
6    // ...
7    if totalShares == 0 {
8      return shim.Error("No shares found")
9    }
10
11   dividendPerShare := dividend / totalShares
12   // ...
13 }
```

Fig. 11: Integer overflow in *gpt-4/153.go*.

```
1  func (s *SmartContract)
     applyForMortgage(APIstub
     shim.ChaincodeStubInterface, args
     []string) sc.Response {
2  // ...
3    income, _ := strconv.Atoi(args[1])
4    loanAmount, _ := strconv.Atoi(args[2])
5    var status string
6    if income/loanAmount < 2 {
7      status = "Rejected"
8    } else {
9      status = "Approved"
10   }
11 // ...
12 }
```

Fig. 12: Possible division by zero in *gpt-4/137.go*.

is used as target channel by the cross-contract invocation `InvokeChaincode` at lines 8-9. The main problem is that `CrossChainBridge.ChannelName` can be change over the time depending to the transaction input. Hence, the target channel can be changed after the code deployment. It leads leads to two issues: (i) it keeps the same transaction context and (ii) it could be lead to missing commitment of write operations in world ledger state of the called contract. In the first case, if the target channel is different from the deployment channel of callee contract, no data about CCI execution is recorded in the target channel, leading to a lack of transparency in blockchain operations and to potential privacy and security risks. In the second case, if the target channel is different and the invoked contract executes write-state operations, these will not affect the channel's ledger and the code will not update data within the world state.

Regarding verification tools, Revive^CC does not provide checks for the detection of cross-channel issues. Instead, GoLiSA detects both transparency issue and missing commitment issues. In particular, GoLiSA implements the TARSIS domain [71], [72] to approximate the possible string values of channels. Then, after the string analysis computation, the analyzer inspects the position of cross-contract operations and, in case, triggers the alarms.

7) Numerical Overflows: The code in Figure 11, taken from the contracts generated by GPT-4, exhibits a possible numerical overflow. In line 5, the method `distributeDividends` collects the total share summing the values of shareholders. Then, it performs a check at line 7 to avoid division by zero at line 11, where it is computed the value of dividend per share. In this contract, an *integer overflow* may occur at line 5, and the code does not have any check to prevent the problem. In fact, if the sum of the shares were to exceed the limit of the integer types, the value would become negative, and the consequence would be negative dividends that remove value rather than add it. In this case, syntactic checks cannot help in detecting overflow problems since they ignore program semantics. Then, GoLiSA is the only tool at our disposal to detect these issues, performing numerical analyses based on semantics abstraction to approximate the possible integer values of variables. In particular, GoLiSA implements the interval domain [58], [73] as a value domain in order to track the abstract values of variables. In this way, after the analysis computation, it is possible to build a semantic check to inspect the program points where arithmetical operations occur, issuing alarms if they can lead to overflows or underflows.

We also conducted an in-depth manual investigation to ensure the presence of checks to prevent overflows/underflows within the generated contracts. However, **regardless of the LLM, there are no guards, checks, or prevention mechanisms relating to numerical overflows/underflows in the generated smart contracts**.

8) Divisions By Zero: Consider the smart contract snippet in Figure 12, generated by GPT-4. At lines 4, it retrieves an arbitrary string value from `args[2]` that is converted in integer value by the function `strconv.Atoi`. Being statically unknown, this value can also be zero at runtime. Hence, the expression `income/loanAmount` at line 6 could be lead to a division by zero, not allowing the user to approve or reject a mortgage, i.e. the `applyForMortgage` method cannot fulfill its purpose.

The division by zero is a classic arithmetic issue that can lead to unexpected results or execution failures. Currently, the Go compiler (Go 1.22) can syntactically detect the direct division by zero. For instance, the expression `1/0` produces an error at compile time. However, the compiler is not able to detect cases where the evaluation of a divisor may yield the value zero. For instance, the expression `strconv.Atoi(args[2])` does not produce any compilation errors. In these cases, the compiler does not consider the program semantics of functions and methods, and then the division by zero is handled at runtime, generating a *panic* error because something unexpected occurs, and the program cannot proceed safely. In the same way, Revive^CC is not able to detect the issue reported in Figure 12 because supports only some syntactic checks for the detection of divisions by zero. As in the case of numerical overflows, it is necessary to reason about program semantics to compute the possible numerical values of a divisor in non-trivial contracts. Instead, GoLiSA can infer the possible intervals of numerical values related to divisors and check whether the value zero is included in them.

VI. Related Work

In this section, we first discuss related work regard the usage of LLMs for code synthesis and software enginerring in general followed by approaches for static analysis of smart contracts. Then, we address work dealing with the security of LLM-generated code and finally, studies that address the intersection of LLMs and the security of smart contracts.

A. LLMs for software engineering

LLMs have profoundly influenced the software engineering landscape [16], [74]. These models, characterized by diverse architectural frameworks and objective functions, have been instrumental in advancing code comprehension and generation tasks [12], [14]. Among the noteworthy instances are Deepseek-Coder [44], CodeLlama [12], and CodeGen2.5 [13], purposefully crafted to address multifaceted challenges encompassing code classification, retrieval, translation, repair, and summarization. These models generate functionally correct programs based on user-provided prompts [15], enhancing developers' productivity [16].

B. Static Analysis of Smart Contracts

In this study, we focus on harnessing the capabilities of LLMs for the explicit purpose of smart contract generation, with a subsequent focus on rigorously testing these contracts for vulnerabilities. Section IV-B discusses what tools are available for Hyperledger Fabric smart contracts and justifies how we chose the static analyzers used in our experiments. A relevant research effort has been focused on the static analysis of Ethereum smart contracts [75], [76]. Our work focuses on HF because it is de facto the standard for enterprise blockchain platforms [24].

C. Security of LLM-generated code

Recently, several approaches have focused on the security of LLM-generated code. He and Vechev [19] assessed the security of such code through adversarial testing and then guided program generation to generate secure or unsafe code. Even if using a different approach based on dynamic analysis, the security assessment of state-of-the-art LLM-generated code resulted in statistics similar to ours (40.9% unsafe code). Similarly, Pearce *et al.* [77] concluded by manual inspection that about 40% of the code generated by GitHub Copilot contains relevant vulnerabilities. Khoury *et al.* [78] conducted a similar analysis to assess the security of code generated by ChatGPT. Further studies [79] consolidated these statistics for various LLMs. On the one hand, our work (and in particular **RQ1**) further consolidates this result, specifically focusing on HF smart contracts. On the other hand, we take a step further by comparing different LLMs (**RQ2**) and by applying and comparing different static analyzers (**RQ3**) instead of dynamic analyses or manual inspection.

D. Intersection of LLMs and Smart Contract Security

Furthermore, Sun et al. [80] introduced GPTScan, a novel approach that combines generative pretrained transformers (GPT) with program analysis using static analysis to detect logic vulnerabilities in solidity files and contract projects. This method leverages the language understanding capabilities of GPT to identify potential security flaws that traditional analysis tools might overlook. For token contracts they achieve high precision and for large projects acceptable precision. Moreover, the tool PropertyGPT by Liu et. al. [81] addresses the topic of formal verification of Solidity smart contracts using retrieval augmented generation. The authors find that their tool can detect several real-word vulnerabilities in smart contracts. Besides vulnerability detection, another approach by Wang et. al. [82], ContractTinker performs automated vulnerability repair on smart contracts written in Solidity using chain-of-thought to break the task down into subtasks. The evaluation shows a high success rate. To the best of our knowledge, none of these approaches analyze smart contracts in the Go language for Hyperledger Fabric.

VII. Conclusion

In this paper, we applied static analyzers for Hyperledger Fabric smart contracts automatically generated by various LLMs. Our main goal was to assess if the generated code is secure, and if there are noticeable differences between different LLMs and static analyzers. For this reason, we selected state-of-the-art tools with different features (commercial or open source, specific for code generation or generic for LLMs, syntactic or semantic for static analyzers).

Our experimentation shows that the generation of HF smart contracts with LLMs does not seem mature enough for its adoption in industrial contexts, as the generated code contains several security issues (**RQ1**). This applies to both open-source code-specific and commercial general-purpose LLMs (**RQ2**). Static analyzers can help to automatically identify code issues, even if only deep (that is, performing computationally expensive semantic) analyses are able to catch a representative set of issues both from a quantitative and a qualitative perspective (**RQ3**). In future work, we plan to investigate how LLMs can be improved and fine-tuned to produce more secure smart contracts.

Acknowledgement

Work partially supported by SERICS (PE00000014 - CUP H73C2200089001) and iNEST (ECS00000043 – CUP H43C22000540006) projects funded by PNRR NextGeneration EU; and by the EU—NGEU National Sustainable Mobility Center (CN00000023), Italian Ministry of University and Research Decree n. 1033—17/06/2022 (Spoke 10).

References

[1] A. K. Kar and L. Navin, "Diffusion of blockchain in insurance industry: An analysis through the review of academic and trade literature," *Telematics and Informatics*, vol. 58, p. 101532, 2021. [Online]. Available: https://doi.org/10.1016/j.tele.2020.101532

[2] L. Olivieri and L. Pasetto, "Towards compliance of smart contracts with the european union data act," in *CEUR Workshop Proceedings*, vol. 3629, 2024, p. 61 – 66. [Online]. Available: https://ceur-ws.org/Vol-3629/paper10.pdf

[3] L. Olivieri, L. Pasetto, L. Negrini, and P. Ferrara, "European union data act and blockchain technology: Challenges and new directions," in *CEUR Workshop Proceedings*, vol. 3791, 2024. [Online]. Available: https://ceur-ws.org/Vol-3791/paper30.pdf

[4] J. Al-Jaroodi and N. Mohamed, "Blockchain in industries: A survey," *IEEE Access*, vol. 7, pp. 36 500–36 515, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2903554

[5] B. Acharya, M. Saad, A. E. Cinà, L. Schonherr, H. D. Nguyen, A. Oest, P. Vadrevu, and T. Holz, "Conning the Crypto Conman: End-to-End Analysis of Cryptocurrency-based Technical Support Scams," in *IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 17–35. [Online]. Available: https://doi.org/10.1109/SP54263.2024.00156

[6] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008, available at https://bitcoin.org/bitcoin.pdf.

[7] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.

[8] N. Popper, "A Hacking of More Than $50 Million Dashes Hopes in the World of Virtual Currency," *The New York Times*, 2016, june 17th.

[9] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020. [Online]. Available: https://doi.org/10.5555/3495724.3495883

[10] J. e. a. Schulman, "Chatgpt: Optimizing language models for dialogue, https://openai.com/blog/chatgpt," 2024.

[11] D. Beste, G. Menguy, H. Hajipour, M. Fritz, A. E. Cinà, S. Bardin, T. Holz, T. Eisenhofer, and L. Schönherr, "Exploring the potential of llms for code deobfuscation," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2025, pp. 267–286. [Online]. Available: https://doi.org/10.1007/978-3-031-97620-9_15

[12] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv:2308.12950*, 2023.

[13] "CodeGen2.5: Small, but mighty — blog.salesforceairesearch.com," https://blog.salesforceairesearch.com/codegen25/, (Accessed 01/2024).

[14] T. Dohmke, "Github copilot is generally available to all developers," Jun. 2022, https://github.blog/2022-06-21-github-copilot-is-generally-available-to-all-developers/ (Accessed 02/2024).

[15] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, and Others, "Evaluating large language models trained on code," *arXiv:2107.03374*, 2021.

[16] A. Ziegler, E. Kalliamvakou, S. Simister, G. Sittampalam, A. Li, A. S. Rice, D. Rifkin, and E. Aftandilian, "Productivity assessment of neural code completion," *6th ACM SIGPLAN International Symposium on Machine Programming*, 2022. [Online]. Available: https://doi.org/10.1145/3520312.3534864

[17] M. Tabachnyk, "Ml-enhanced code completion improves developer productivity." [Online]. Available: https://blog.research.google/2022/07/ml-enhanced-code-completion-improves.html?m=1

[18] S. Imai, "Is github copilot a substitute for human pair-programming? an empirical study," *IEEE/ACM 44th International Conference on Software Engineering*, pp. 319–321, 2022. [Online]. Available: https://doi.org/10.1145/3510454.3522684

[19] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *ACM SIGSAC Conference on Computer and Communications Security*, 2023, p. 1865–1879.

[20] M. A. Mahdi H. Miraz, "Blockchain Enabled Smart Contract Based Applications: Deficiencies with the Software Development Life Cycle Models," *Baltica Journal*, vol. 33, pp. 101–116, 2020.

[21] A. Bosu, A. Iqbal, R. Shahriyar, and P. Chakraborty, "Understanding the motivations, challenges and needs of Blockchain software developers: a survey," *Empir. Softw. Eng.*, vol. 24, no. 4, pp. 2636–2673, 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09708-7

[22] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, "Blockchain-Oriented Software Engineering: Challenges and New Directions," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 169–171. [Online]. Available: https://doi.org/10.1109/icse-c.2017.142

[23] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190538

[24] IBM, "What is hyperledger fabric?" 2024, https://www.ibm.com/topics/hyperledger (Accessed 01/2024).

[25] ——, "Hyperledger achieves huge milestone: Introducing Hyperledger Fabric 2.0," 2023, https://www.ibm.com/blog/hyperledger-achieves-huge-milestone-introducing-hyperledger-fabric-2-0/ (Accessed 01/2024).

[26] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *IEEE International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1. IEEE, 2016, pp. 470–481. [Online]. Available: https://doi.org/10.1109/SANER.2016.105

[27] X. Rival and K. Yi, *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020.

[28] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," *ACM Comput. Surv.*, vol. 54, no. 7, 2021. [Online]. Available: https://doi.org/10.1145/3464421

[29] S. Kim and S. Ryu, "Analysis of blockchain smart contracts: Techniques and insights," in *IEEE Secure Development*, 2020, pp. 65–73. [Online]. Available: https://doi.org/10.1109/SecDev45635.2020.00026

[30] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, "Verification of smart contracts: A survey," *Pervasive and Mobile Computing*, vol. 67, p. 101227, 2020. [Online]. Available: https://doi.org/10.1016/j.pmcj.2020.101227

[31] P. Lv, Y. Wang, Y. Wang, and Q. Zhou, "Potential Risk Detection System of Hyperledger Fabric Smart Contract based on Static Analysis," in *IEEE Symposium on Computers and Communications, ISCC 2021, Athens, Greece, September 5-8, 2021*. IEEE, 2021, pp. 1–7. [Online]. Available: https://doi.org/10.1109/ISCC53001.2021.9631249

[32] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun, "Potential risks of hyperledger fabric smart contracts," in *IEEE International Workshop on Blockchain Oriented Software Engineering*, 2019, pp. 1–10. [Online]. Available: https://doi.org/10.1109/IWBOSE.2019.8666486

[33] C. Paulsen, "Revisiting smart contract vulnerabilities in hyperledger fabric," 2021.

[34] L. Olivieri and F. Spoto, "Software verification challenges in the blockchain ecosystem," *International Journal on Software Tools for Technology Transfer*, 2024, published 2024/07/12. [Online]. Available: https://doi.org/10.1007/s10009-024-00758-x

[35] 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.

[36] B. Acharya, D. Lazzaro, E. López-Morales, A. Oest, M. Saad, A. E. Cinà, L. Schönherr, and T. Holz, "The imitation game: exploring brand impersonation attacks on social media platforms," in *Proceedings of the 33rd USENIX Conference on Security Symposium*, 2024.

[37] B. Acharya, D. Lazzaro, A. E. Cinà, and T. Holz, "Pirates of charity: Exploring donation-based abuses in social media platforms," in *Proceedings of the ACM on Web Conference 2025*, 2025, pp. 3968–3981. [Online]. Available: https://doi.org/10.1145/3696410.3714634

[38] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, "Repair is nearly generation: Multilingual program repair with llms," in *AAAI Conference on Artificial Intelligence*, 2023. [Online]. Available: https://doi.org/10.1609/aaai.v37i4.25642

[39] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023. [Online]. Available: https://doi.org/10.1145/3611643.3616271

[40] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *arXiv::2204.02311*, 2022.

[41] C. Zhang, C. Zhang, S. Zheng *et al.*, "A complete survey on generative ai (aigc): Is chatgpt from gpt-4 to gpt-5 all you need?" 2023.

[42] W. Zeng, X. Ren, T. Su, H. Wang, Y. Liao, Z. Wang, X. Jiang, Z. Yang, K. Wang, X. Zhang *et al.*, "Pangu-α: Large-scale autoregressive

pretrained chinese language models with auto-parallel computation," *arXiv:2104.12369*, 2021.

[43] S. Wu, X. Zhao, T. Yu, R. Zhang, C. Shen, H. Liu, F. Li, H. Zhu, J. Luo, L. Xu *et al.*, "Yuan 1.0: Large-scale pre-trained language model in zero-shot and few-shot learning," *arXiv:2110.04725*, 2021.

[44] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv:2401.14196*, 2024.

[45] Anthropic, "Claude-3.5, https://www.anthropic.com/news/claude-3-5-sonnet."

[46] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv:2303.08774*, 2023.

[47] L. Olivieri, V. Arceri, B. Chachar, L. Negrini, F. Tagliaferro, F. Spoto, P. Ferrara, and A. Cortesi, "General-purpose languages for blockchain smart contracts development: A comprehensive study," *IEEE Access*, vol. 12, pp. 166 855–166 869, 2024. [Online]. Available: https://doi.org/10.1109/ACCESS.2024.3495535

[48] Go language project, "gofmt - gofmt formats go program," 2015, https://pkg.go.dev/cmd/gofmt (Accessed 01/2024).

[49] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," *arXiv:1904.09751*, 2019.

[50] Albert Danial, "cloc - count lines of code," 2015, https://github.com/AlDanial/cloc (Accessed 01/2024).

[51] G. A. Oliva, A. E. Hassan, and Z. M. Jiang, "An Exploratory Study of Smart Contracts in the Ethereum Blockchain Platform," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1864–1904, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09796-5

[52] K. Yamashita and J. Ry, "Chaincode Analyzer," 2020, https://github.com/hyperledger-labs/chaincode-analyzer. Accessed 02/2024.

[53] C. Siva, "Revivecc," 2021, https://github.com/sivachokkapu/revive-cc. Accessed 02/2024.

[54] Minko Gechev, "Revive analizer," 2019, https://github.com/mgechev/revive (Accessed 01/2024).

[55] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Network and Distributed System Security Symposium*, 2018.

[56] L. Olivieri, V. Arceri, L. Negrini *et al.*, "GoLiSA - GitHub Repository," 2020, https://github.com/lisa-analyzer/go-lisa (Accessed 01/2024).

[57] L. Olivieri, L. Negrini, V. Arceri, F. Tagliaferro, P. Ferrara, A. Cortesi, and F. Spoto, "Information Flow Analysis for Detecting Non-Determinism in Blockchain," in *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, ser. Leibniz International Proceedings in Informatics, vol. 263. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 23:1–23:25. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2023.23

[58] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *ACM Symposium on Principles of Programming Languages*. ACM, 1977, pp. 238–252.

[59] ——, "Systematic Design of Program Analysis Frameworks," in *6th Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1979, pp. 269–282.

[60] L. Olivieri, F. Tagliaferro, V. Arceri, M. Ruaro, L. Negrini, A. Cortesi, P. Ferrara, F. Spoto, and E. Talin, "Ensuring determinism in blockchain software with golisa: an industrial experience report," in *ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, 2022, p. 23–29. [Online]. Available: https://doi.org/10.1145/3520313.3534658

[61] A. S. Tanenbaum, *Distributed systems principles and paradigms*, 2007.

[62] L. Olivieri, L. Negrini, V. Arceri, P. Ferrara, and A. Cortesi, "Detection of read-write issues in hyperledger fabric smart contracts," in *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*. Association for Computing Machinery, 2025, p. 329–337. [Online]. Available: https://doi.org/10.1145/3672608.3707721

[63] L. Olivieri, L. Negrini, V. Arceri, B. Chachar, P. Ferrara, and A. Cortesi, "Detection of phantom reads in hyperledger fabric," *IEEE Access*, vol. 12, pp. 80 687–80 697, 2024. [Online]. Available: https://doi.org/10.1109/ACCESS.2024.3410019

[64] S. K. Shabna Madathil Thattantavida, "IBM Developer - Learn best practices for debugging and error handling in an enterprise-grade blockchain application," 2023, https://developer.ibm.com/blogs/debugging-and-error-handling-best-practices-in-a-blockchain-application (Accessed 01/2024).

[65] L. Olivieri, L. Negrini, V. Arceri, P. Ferrara, A. Cortesi, and F. Spoto, "Static detection of untrusted cross-contract invocations in go smart contracts," in *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '25, 2025, p. 338–347. [Online]. Available: https://doi.org/10.1145/3672608.3707728

[66] L. Olivieri, T. Jensen, L. Negrini, and F. Spoto, "Michelsonlisa: A static analyzer for tezos," in *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2023, pp. 80–85. [Online]. Available: https://doi.org/10.1109/PerComWorkshops56833.2023.10150247

[67] L. Olivieri, L. Negrini, V. Arceri, T. Jensen, and F. Spoto, "Design and implementation of static analyses for tezos smart contracts," *Distrib. Ledger Technol.*, vol. 4, no. 2, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3643567

[68] Google, "The Go Programming language Specification - Integer Overflow." [Online]. Available: https://go.dev/ref/spec#Integer_overflow

[69] T. Min and W. Cai, "A security case study for blockchain games," in *2019 IEEE Games, Entertainment, Media Conference*, 2019, pp. 1–8. [Online]. Available: https://doi.org/10.1109/GEM.2019.8811555

[70] NIST, "NVD - CVE-2018-10299 Detail," https://nvd.nist.gov/vuln/detail/cve-2018-10299 (Accessed 01/2024).

[71] L. Negrini, V. Arceri, A. Cortesi, and P. Ferrara, "Tarsis: An effective automata-based abstract domain for string analysis," *Journal of Software: Evolution and Process*, vol. 36, no. 8, p. e2647, 2024. [Online]. Available: https://doi.org/10.1002/smr.2647

[72] L. Negrini, V. Arceri, P. Ferrara, and A. Cortesi, "Twinning automata and regular expressions for string static analysis," in *Verification, Model Checking, and Abstract Interpretation*, 2021. [Online]. Available: https://doi.org/10.1007/978-3-030-67067-2_13

[73] L. Negrini, V. Arceri, L. Olivieri, A. Cortesi, and P. Ferrara, "Teaching through practice: Advanced static analysis with lisa," in *Formal Methods Teaching*. Cham: Springer Nature Switzerland, 2024, pp. 43–57. [Online]. Available: https://doi.org/10.1007/978-3-031-71379-8_3

[74] T. Dohmke, "Github copilot x: The ai-powered developer experience," Sep 2023. [Online]. Available: https://github.blog/2023-03-22-github-copilot-x-the-ai-powered-developer-experience/

[75] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *1st international workshop on emerging trends in software engineering for blockchain*, 2018, pp. 9–16.

[76] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of ethereum smart contracts," in *Computer Aided Verification: 30th International Conference*, 2018, pp. 51–78. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3_4

[77] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *IEEE Symposium on Security and Privacy*, 2022, pp. 754–768. [Online]. Available: https://doi.org/10.1109/SP46214.2022.9833571

[78] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by chatgpt?" in *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2023, pp. 2445–2451. [Online]. Available: https://doi.org/10.1109/SMC53992.2023.10394237

[79] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv:2305.06161*, 2023.

[80] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *IEEE/ACM International Conference on Software Engineering*, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639117

[81] 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:2405.02580*, 2024.

[82] C. Wang, J. Zhang, J. Gao, L. Xia, Z. Guan, and Z. Chen, "Contracttinker: Llm-empowered vulnerability repair for real-world smart contracts," in *IEEE/ACM International Conference on Automated Software Engineering*, 2024, p. 2350–2353. [Online]. Available: https://doi.org/10.1145/3691620.3695349