Vulnerability Assessment of LLM-Generated Smart Contracts in Ethereum

Mirco Vella[1], Antonio Emanuele Cinà[1,*], Marina Ribaudo[1] and Fabio Roli[1,2]

[1]DIBRIS, Università di Genova, Via Dodecaneso, 3, Genova, Italy
[2]DIEE, Università di Cagliari, Via Marengo, 09123 Cagliari, Italy

Abstract

Blockchain technology and smart contracts are increasingly used in finance, government, and industry. At the same time, Large Language Models (LLMs) offer new possibilities, such as generating smart contracts from natural language, which can lower costs and speed up development. However, smart contracts are often vulnerable to security flaws, whether written by humans or AI, leading to significant financial losses. This study systematically evaluates the quality and security of smart contract code generated by LLMs in the Ethereum blockchain ecosystem. We generated 250 smart contracts using two state-of-the-art models, GPT-4 and DeepSeek-Coder, and assessed their security using automated vulnerability detection tools, Slither and Mythril. Our findings reveal that while LLM-generated smart contracts exhibit improvements in syntactic correctness and coherence, they still suffer from critical security vulnerabilities, making them unsuitable for fully autonomous development.

Keywords

Large Language Models, Smart Contract Generation, Auditing Tools, Program Verification, Security

1. Introduction

The integration of Large Language Models (LLMs) into software development remains an active area of research, driven by the rapid emergence of diverse models with distinct capabilities [1, 2]. These models demonstrate significant potential in automating various programming tasks, streamlining software development through the generation of boilerplate code [3], function completion from minimal input [4], and assistance in debugging [5]. Reducing the time required for coding and troubleshooting allows developers to focus on strategic and creative aspects of their work [6, 7].

One particularly promising yet challenging application of LLMs is smart contract development for blockchain systems. As a core component of programmable blockchain platforms like Ethereum [8], smart contracts enable the automated execution of agreements [9]. However, a major concern when using LLMs to generate code is their lack of awareness of security risks. Studies have shown that AI-generated code can introduce vulnerabilities that compromise security standards [10]. In the case of smart contracts, such flaws can lead to severe financial losses and security breaches [11, 12]. As a consequence, the use of LLMs for smart contract development requires careful evaluation to determine whether AI-generated code meets necessary security and reliability standards. In this regard, auditing tools offer a valuable means of assessing the security of the code, and this paper aims to address the following research question:

RQ. *"Can LLMs accelerate smart contract development while ensuring high-quality, secure code?"*

To explore this question, we used a dataset of diverse prompts for smart contract generation which were processed by two AI models, producing code that was subsequently analyzed using two auditing tools. The findings provide insights into whether LLM-generated smart contracts are secure and reliable.

GeCoIn 2025: Generative Code Intelligence Workshop, co-located with the 28th European Conference on Artificial Intelligence (ECAI-2025), October 26, 2025 — Bologna, Italy

*Corresponding author.
vellamirco@libero.it (M. Vella); antonio.cina@unige.it (A. E. Cinà); marina.ribaudo@unige.it (M. Ribaudo); fabio.roli@unige.it (F. Roli)
0000-0003-3807-6417 (A. E. Cinà); 0000-0003-0697-2225 (M. Ribaudo); 0000-0003-4103-9190 (F. Roli)

The paper is structured as follows. Section 2 provides background information on blockchain technology, with a focus on the Ethereum platform, and introduces the notion of LLMs. Furthermore, it also reviews relevant literature on AI-assisted smart contract development. Section 3 details the data collection process used to construct the dataset for evaluating the selected LLMs and the methodology for generating smart contracts automatically. Section 4 presents the experimental results, assessing key metrics such as contract compilability and vulnerability detection. Finally, Section 5 summarizes the findings and discusses potential directions for future research.

2. Preliminaries and Related Work

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

2.1. Smart contracts vulnerabilities

Blockchain is a decentralized architecture designed to securely record transactions and data in an immutable ledger structured as a chain of blocks, maintained through consensus among peers. Beyond basic data storage, many blockchain platforms, such as Ethereum [8], support the execution of smart contracts, which are self-executing programs that enforce agreements between parties without the need for intermediaries. First envisioned by Szabo [13], smart contracts enable the development of decentralized trusted applications known as dApps [14].

Despite their advantages, smart contracts are susceptible to critical vulnerabilities[1] that can compromise their functionality and security [16, 17]. Examples include exceeding the *gas limit*, causing denial-of-service, miner manipulation of `block.timestamp`, and *reentrancy* attacks, such as the DAO hack [18]. These vulnerabilities underscore the necessity for systematic development and rigorous security auditing to ensure robust, reliable smart contracts and minimize financial risks [19]. Aligned with this need, Destefanis et al. [19] stress adopting strong software engineering practices for creating a dedicated discipline for smart contract development, citing major incidents like the Parity wallet attack [20] that resulted in significant losses. Kim and Ryu [21] classify security methods into static analysis for vulnerability detection and correctness, plus dynamic analysis, identifying key challenges and future directions. Sendner et al. [22] evaluate automated tools like Slither [23, 24] and Mythril [25, 26], concluding that combining multiple scanners is essential for effective vulnerability detection.

2.2. Large Language Models

Large Language Models (LLMs) leverage transformer architectures [27] for diverse language processing tasks. Trained on extensive datasets of natural language and code [28], they develop strong language understanding, generation, and in-context learning capabilities. LLMs have become essential tools across multiple fields, enabling applications such as scam detection [29, 30, 31], automated code generation [32], code repair [33, 34], and reverse deobfuscation [35]. Their integration into software engineering has notably accelerated code synthesis and automation workflows [6, 7], including the generation of smart contracts for blockchain platforms. Given their growing relevance and production for code synthesis, researchers have examined the security of LLM-generated code more broadly. He and Vechev [10] evaluate security risks through adversarial testing, guiding LLMs to generate both secure and intentionally unsafe code. Similarly, Pearce et al. [36] manually inspect GitHub Copilot-generated code, finding that approximately 40% contains security vulnerabilities. Khoury et al. [37] extend this analysis to ChatGPT-generated code, corroborating similar risk levels. More recently, Li et al. [38] provide a comparative assessment across multiple LLMs, reinforcing these findings. In the context of smart contract generation via LLMs, Karanjai et al. [39] compare Google PaLM2 and GPT-3.5 in generating Solidity smart contracts from natural language descriptions. While PaLM2 achieved

[1]The OWASP website [15] lists the top 10 vulnerabilities.

higher accuracy, both models frequently produced subtle bugs and exhibited poor coding practices, highlighting concerns about their reliability for production deployment. Similarly, Napoli et al. [40] evaluated ChatGPT's ability to identify and fix vulnerabilities in smart contracts, finding a success rate of 57.1% after multiple attempts. They conclude that while LLMs can significantly assist developers, they cannot fully replace expert human oversight, particularly for security-critical tasks. Barbàra et al. [41] assess GPT-4's ability to autonomously generate Solidity smart contracts from legal documents using multiple prompt variants. Their evaluation, limited to 80 lease-agreement contracts and relying solely on Slither for analysis, reveals that GPT-4 struggles with producing production-ready code due to subtle bugs and inconsistencies between prompts and outputs. However, the study does not incorporate advanced symbolic execution analysis tools (e.g., Mythril [25]), which limits the depth of vulnerability detection and prevents classification of issues by severity. Olivieri et al. [42] extend this line of work to Hyperledger Fabric [43, 44], evaluating LLMs' ability to generate secure smart contracts in Go from natural language prompts. Their results reveal that while LLMs can accelerate development, the generated contracts often require substantial debugging and manual intervention due to quality and security deficiencies.

Our work builds upon these efforts by conducting a security assessment of LLM-generated smart contracts. We extend prior research by (1) focusing on Ethereum-based contracts, (2) comparing both commercial and open-source state-of-the-art LLMs for code synthesis, and (3) evaluating the effectiveness of multiple static analysis tools, rather than relying solely on dynamic analysis or manual review. Lastly, complementary to prior studies, our analysis not only identifies the vulnerabilities but also classifies their severity, revealing the critical risks posed by certain flaws in the generated contracts.

3. Data Collection, Contract Generation and Analysis

This section outlines the methodology (illustrated in Figure 1) employed for generating smart contracts using LLMs and details the experimental workflow designed to evaluate their performance and security.[2]

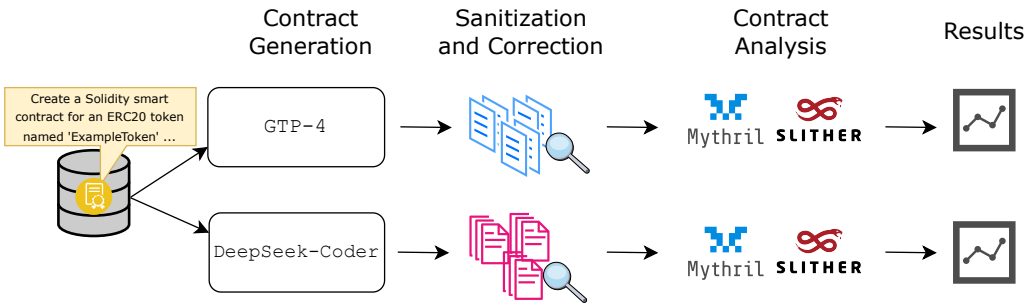

Figure 1: Pipeline of the workflow adopted in this study.

3.1. Dataset construction

In our context, a prompt is the natural language input that guides an LLM to generate functional smart contract code [45]. To ensure reproducibility and enable systematic evaluation, a well-structured *prompt dataset* is essential. For this study, we adopt a dataset consisting of predefined, structured prompts that explicitly instruct the model to generate smart contracts from natural language descriptions. We selected an external prompt dataset [46] hosted on Hugging Face [47, 48], a prominent repository for machine learning resources.

To better understand the provenance and the structure of the dataset, we consulted its creators. According to their documentation, the dataset was created using the Meta LLaMA 3 8B Instruct

[2]For the code and the chosen prompts refer to https://github.com/vmirco/Ethereum-Smart-Contract-Generation-and-Analysis.

model [49], which processed an initial corpus [50] of Solidity smart contracts sourced from public GitHub repositories. For each contract, the model generated three distinct prompt levels—Beginner, Average, Expert—with progressively increasing complexity. Listing 1 presents an example from a single entry in the dataset, illustrating one prompt at the Average level, which is the one adopted in our study.

```
""" Create a smart contract that builds upon the WETHOmnibridgeRouter contract by adding
    features for account registration, token wrapping, and relay. The contract should
    integrate with the Omnibridge and WETH contracts. Include methods for registering
    and wrapping tokens, as well as functionality for relaying tokens to specific
    recipients. The contract should emit events upon successful token wrapping and
    relaying. Consider implementing error handling and validation checks for user input.
```

Listing 1: Average level prompt example.

Contract Generation: Instructions for the LLMs. We designed prompt templates simulating typical user inputs via graphical interfaces, specifying contract features without detailed implementation instructions. This approach evaluates the ability of LLMs to translate natural language specifications into coherent, deployable Solidity code. For instance, Listing 2 shows a prompt for generating an ERC20 token contract.

```
""" Create a Solidity smart contract for an ERC20 token named 'ExampleToken' ('EXT')
    with a supply of 1,000,000 tokens. Implement transfer, approve, and transferFrom
    functions. Include owner-only minting and token burning functionalities.
```

Listing 2: Prompt example for ERC20 tokens.

To ensure consistent and deployable output, prompts explicitly require Solidity code compatible with version `0.8.0`, as shown in Listing 3. This fixed version avoids discrepancies in compilation and analysis, eliminating the need for manual adjustments across different compiler versions.

```
""" You will generate deployable smart contract code in Solidity based on the prompt I
    provide. Use Solidity version ^0.8.0.
```

Listing 3: Instruction template.

In practice, LLMs outputs sometimes included extraneous text (e.g., introductory phrases, `import` statements, or markdown syntax) that broke compilation. We therefore refined the instructions to restrict output to pure Solidity code and to replace `import` directives with inlined code to avoid unresolved dependencies (Listing 4).

```
""" The output should contain only Solidity code – no comments or markdown such as "```
    sol". I should be able to copy your response and paste it in a sol file to deploy.
    Do not use import statement, only code, if there's any import, replace it with code
    for the actual imported contract.
```

Listing 4: Instruction template (cnt).

3.2. Contract Analysis: Slither and Mythril

To identify potential vulnerabilities in generated smart contracts, we employed two state-of-the-art tools: Slither [23, 24] and Mythril [25, 26]. These tools complement each other by combining static and symbolic analysis techniques, enabling a thorough security assessment.

Slither. Slither is a Python-based static analyzer for Solidity and Vyper that uses an intermediate representation to detect vulnerabilities. It provides severity classifications for findings: *Informational*, *Optimization*, *Low*, *Medium*, and *High*. The first two categories offer insights and recommendations for code quality and gas efficiency but do not necessarily indicate security flaws. The latter three denote increasing severity levels of security risks, helping prioritize remediation efforts. For each contract, we ran Slither analysis and saved its output. If compilation errors or warnings were reported, the analysis for that contract was halted. Otherwise, identified vulnerabilities were classified and aggregated to quantify the overall security risk.

Mythril. Mythril performs symbolic execution on Ethereum Virtual Machine bytecode, systematically exploring execution paths to detect vulnerabilities. Given the high computational cost of symbolic analysis, we limited Mythril's runtime to 20 minutes per contract.[3] Mythril outputs vulnerabilities categorized by severity levels (*Low*, *Medium*, and *High*) and suggests potential mitigations. Our analysis proceeded in two phases: first, vulnerabilities were grouped by severity to evaluate criticality; second, they were classified by type to identify the most common security threats.

4. Results

This section presents the results of the comparative analysis of 250 Solidity smart contracts generated using GPT-4 and DeepSeek-Coder. The prompts used for code generation were randomly selected from the initial dataset, focusing on the Average prompt level. The primary objective of this analysis is to evaluate the quality of the generated contracts in terms of errors, vulnerabilities, and adherence to blockchain coding best practices.

4.1. Code Compilation Success Rate

A key indicator of smart contract reliability is whether the code compiles successfully. Compilation failures render contracts unusable and typically result from either syntactic errors or missing dependencies. The latter often stems from the use of `import` statements that reference unavailable external libraries. To reduce this issue, we explicitly instructed both models to avoid using `import` statements (see Listing 4). Nonetheless, some generated contracts included such statements, leading to failed analyses by Slither, which effectively detects missing dependencies and syntax issues.

Our results reveal a significant performance gap between the two models. Using Slither, we found that DeepSeek-Coder produced only 46 non-compilable contracts (18% of its total), while GPT-4 failed to compile 139 contracts (55%). Focusing on import-related errors, GPT-4 generated 14 contracts with forbidden `import` statements, compared to just 2 in the DeepSeek-Coder set. These findings suggest two key differences. First, DeepSeek-Coder is more capable of generating syntactically correct Solidity code, likely due to its specialization in code generation. Second, DeepSeek-Coder demonstrates better adherence to prompt instructions. After being asked to avoid imports, it generally embedded the required logic directly into the code. In contrast, GPT-4 often ignored the instruction and continued to use `import` statements. Although import-related errors are easily fixed in a real development context, their presence highlights differences in model behavior. Developers typically have access to standard libraries, but models that rely on unavailable imports during generation are less robust in isolated or constrained environments.

To further assess potential usability of LLMs code smart contracts synthesis, we manually corrected contracts with minor syntax issues or import errors. After this intervention, the non-compilation

[3] Analyzing 250 contracts required approximately 80 hours of cumulative runtime, balancing thoroughness and efficiency.

Table 1
Vulnerability types detected by Mythril and Slither in compilable vulnerable contracts.

	Mythril			Slither				
	Low	Medium	High	Low	Medium	High	Informational	Optimization
GPT-4	60.94	28.12	10.94	10.19	3.60	2.30	71.02	12.89
DeepSeek-Coder	53.08	37.69	9.23	9.80	2.20	1.00	65.30	21.70

rate for GPT-4 dropped to 35.74%, and for DeepSeek-Coder to 15.52%, confirming the latter's superior reliability in generating compilable smart contracts.

4.2. Average Vulnerabilities per Type

After sanitizing the smart contracts generated by the two LLMs, we analyzed the number of vulnerabilities detected by Slither. Among all contracts generated by GPT-4, 35.74% were free of vulnerabilities, while 28.52% contained at least one security issue. The remaining 35% were excluded from this analysis due to compilation failure. In practice, this means that approximately one out of three contracts produced by GPT-4 compiled successfully and was free of vulnerabilities. For DeepSeek-Coder, among the 84.48% of contracts that compiled successfully, 68.96% were found to contain vulnerabilities. Despite being optimized for code generation, this indicates that roughly two-thirds of DeepSeek-Coder's compilable contracts were still insecure.

To further understand the nature of these issues, we analyzed the distribution of vulnerability severities within the subset of compilable and vulnerable contracts, using both Mythril and Slither. As shown in Table 1, both models exhibited a consistent pattern: *Low*-severity vulnerabilities appeared most frequently, followed by *Medium* and then *High* severity issues. Lastly, we observe a significant difference in the output between the two tools. Slither reports additional categories, such as *Informational* and *Optimization*, which collectively account for over 80% of its findings, as reflected in the last two columns of the table. The *Informational* category highlights general coding issues or patterns that may affect readability or maintainability, while *Optimization* refers to gas-inefficient code that, although not directly insecure, could lead to increased execution costs on-chain.

4.3. Average Vulnerabilities per Contract

In this section we provide an additional analysis conducted with Slither and Mythril focusing on the average number of vulnerabilities detected per contract.

Using Slither. Both models yield comparable results, with GPT-4 showing a slight advantage when considering all vulnerability categories. The averages are calculated from the subset of vulnerable contracts and thus do not represent the entire set of generated contracts. An average of seven vulnerabilities per contract was found and it may seem significant, but it is important to notice that this figure drops to approximately one when the *Informational* and *Optimization* categories are excluded. In this refined evaluation, DeepSeek-Coder exhibits a slight advantage.

Using Mythril. Before interpreting these results, it is important to recall that Mythril was executed with a processing time constraint of 20 minutes per contract. This limitation may have affected the accuracy of the findings compared to the full capabilities of the tool. Table 2 presents the vulnerabilities identified by Mythril, classified according to their SWC (Smart Contract Weakness Classification) tags [51]. Although GPT-4 produced a higher number of non-compilable contracts, most of its compilable contracts were secure, resulting in 64 detected vulnerabilities. In contrast, contracts generated by DeepSeek-Coder exhibited 130 vulnerabilities. The most prevalent issue was reentrancy (SWC-107), a well-known vulnerability neither model could mitigate effectively, highlighting the challenges LLMs face in understanding the complexities of self-executing functions. Additionally, DeepSeek-Coder showed a higher occurrence of SWC-116, which relates to the use of built-in variables such as `block.number` and `block.timestamp` to trigger time-dependent events. Another notable vulnerability, unprotected

Table 2
Occurrences of SWCs in the generated smart contracts.

ID	Name	GPT-4	DeepSeek-Coder
SWC-104	Unchecked return value	0	1
SWC-105	Unprotected Ether withdrawal	5	6
SWC-107	Reentrancy	34	83
SWC-110	Assert violation	1	1
SWC-112	Delegatecall to Untrusted Callee	2	6
SWC-113	DoS with Failed Call (improper check/handling of conditions)	6	1
SWC-114	Transaction Order Dependence (race condition)	3	3
SWC-115	Authorization through tx.origin (obsolete function)	1	0
SWC-116	Improper use of block.timestamp or block.number	7	27
SWC-120	Weak source of randomness	2	2
SWC-123	Requirement Violation (require() violation)	3	0
		64	**130**

Ether withdrawal (SWC-105), was found in contracts generated by both models at similar rates. This vulnerability is particularly critical, as it allows unauthorized users to withdraw contract funds in an uncontrolled manner, potentially leading to financial loss.

4.4. Contract Code Analysis

In this section, we analyze one representative smart contract generated by each model, starting with DeepSeek-Coder. The prompt provided to both models is shown in Listing 5, and an excerpt of the generated Solidity code is presented in Listing 6.

```
""" Create a smart contract for a
    cryptocurrency-based miner engineer
    game. The contract should allow
    players to buy and sell engineer
    characters, buy boosters for mining,
     and change virus types. The
    contract should have an interface
    for mining war games and mini-games.
     The contract structure should
    include defining structs for miner
    and boost data, mapping player
    information, and implementing
    functions for buying and selling
    engineers. Implement the contract
    functionality by writing Solidity
    code that mirrors the functionality
    of the provided code snippet.
```

Listing 5: Input Prompt.

```
1  contract MinerEngineerGame {
2      mapping(uint256 => Miner) public miners;
3      mapping(uint256 => Booster) public boosters;
4      mapping(address => Player) public players;
5      ...
6      function buyMiner(uint256 _minerId) external {
7          Miner storage miner = miners[_minerId];
8          require(miner.price > 0, "Miner does not
               exist");
9          Player storage player = players[msg.sender];
10         require(player.balance >= miner.price,
11                                  "Not enough balance");
12         player.balance -= miner.price;
13         player.minerIds.push(_minerId);
14         emit MinerBought(msg.sender, _minerId);
15     }
```

Listing 6: Portion of code generated by DeepSeek-Coder.

The code produced by DeepSeek-Coder contains critical issues related to uninitialized variables, specifically the `miners` and `boosters` mappings. These state variables are defined but accessed without prior initialization, as flagged by Slither. In Solidity, reading from an uninitialized mapping entry returns a default-constructed instance. As shown in lines 2 and 3, this behavior may allow a malicious actor to acquire a `Miner` with no cost, since its price defaults to zero. Such a vulnerability could be exploited to gain unfair advantages in application logic, such as mining or trading, without spending any resources. This example highlights the importance of explicitly initializing state variables to ensure correct behavior and prevent unintended vulnerabilities.

By contrast, the code generated by GPT-4 for the same prompt avoids this issue. It enforces a check for sufficient user balance (e.g., at least 1 Ether) before creating a `Miner`, and it initializes the `Miner` instance with explicit attributes. This approach prevents access to default-constructed mappings by ensuring that objects are created only under valid conditions. The solution generated by GPT-4 therefore

reflects better coding practices and more proactive vulnerability prevention.

4.5. Discussion

As shown, both GPT-4 and DeepSeek-Coder were able to generate smart contracts that responded to prompt requirements, though their outputs frequently contained vulnerabilities. GPT-4 demonstrated stronger alignment with complex prompts but produced a higher proportion of non-compilable contracts, mainly due to unresolved `import` statements and syntactic inconsistencies. In contrast, DeepSeek-Coder showed greater syntactic reliability, with fewer than 20% of its contracts failing to compile. However, its outputs contained more vulnerabilities, especially those related to reentrancy (SWC-107) and improper timestamp usage (SWC-116).

Our comparative analysis revealed recurring issues in LLM-generated code, including unchecked return values, flawed authorization mechanisms, and logical errors. While Slither provided broader categories, including *Informational* and *Optimization* insights, Mythril offered more granular detection of specific vulnerability types. Reentrancy, misuse of built-in variables, and unprotected Ether withdrawals were among the most common issues across both models.

In summary, while LLMs exhibit promising capabilities in automating smart contract development, our findings indicate that they are not yet reliable enough to ensure the security and correctness required for deployment in real-world blockchain environments. Revisiting our research question, *"Can LLMs accelerate smart contract development while ensuring high-quality, secure code?"*, the empirical evidence points toward a cautious conclusion. Although LLMs can support early-stage prototyping and reduce development effort, they currently fall short of meeting the standards necessary for producing robust, vulnerability-free code without human oversight.

5. Conclusion

This work investigated the capability of LLMs, namely GPT-4 and DeepSeek-Coder, to autonomously generate secure Solidity smart contracts for the Ethereum blockchain. Using a dataset of predefined prompts and leveraging static and symbolic analysis tools, we evaluated contract correctness, adherence to prompt specifications, and vulnerability presence. Our results suggest that while LLMs can generate syntactically correct and functionally relevant smart contracts, their outputs are not yet sufficiently reliable for fully autonomous deployment. The generated contracts frequently require manual review and correction to address security weaknesses.

This study has some limitations that may affect the generalizability of its results. The dataset of 250 smart contracts may not fully capture the diversity and complexity of real-world contracts. Additionally, the use of predefined prompt templates could influence model outputs and introduce bias. However, the dataset size and methodology remain consistent with or larger than those used in related studies, providing a meaningful basis for comparison and analysis.

Future research should focus on improving the ability of LLMs to handle common smart contract vulnerabilities like reentrancy and to better understand the context of decentralized systems. Using LLMs as supportive tools within development teams, working alongside experts, can enhance their usefulness while reducing potential errors. Since LLM technology is advancing quickly, regular assessment will help guide how these models can be safely and effectively applied in smart contract development.

Acknowledgment

This work was partially supported by project FISA-2023-00128 funded by the MUR program "Fondo italiano per le scienze applicate"; the EU—NGEU National Sustainable Mobility Center (CN00000023), Italian Ministry of University and Research Decree n. 1033—17/06/2022 (Spoke 10); and projects SERICS (PE00000014) and FAIR (PE00000013) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] 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).

[2] S. A. Research, CodeGen2.5: Small, but Mighty, https://blog.salesforceairesearch.com/codegen25/, 2023. Accessed: 2025-07.

[3] M. Schäfer, S. Nadi, A. Eghbali, F. Tip, An empirical evaluation of using large language models for automated unit test generation, IEEE Transactions on Software Engineering (2023).

[4] T. Dohmke, Github copilot is generally available to all developers, 2022. Https://github.blog/2022-06-21-github-copilot-is-generally-available-to-all-developers/ (Accessed: 2025-07).

[5] Y. Majdoub, E. Ben Charrada, Debugging with open-source large language models: An evaluation, in: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2024, pp. 510–516.

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

[7] M. Tabachnyk, ML-enhanced code completion improves developer productivity, 2022. URL: https://blog.research.google/2022/07/ml-enhanced-code-completion-improves.html?m=1.

[8] G. Wood, Ethereum: A Secure Decentralised Generalised Transaction Ledger, Ethereum Project Yellow Paper (2014). URL: https://cryptodeep.ru/doc/paper.pdf.

[9] G. A. Oliva, A. E. Hassan, Z. M. Jiang, An exploratory study of smart contracts in the Ethereum blockchain platform, Empirical Software Engineering 25 (2020) 1864–1904.

[10] J. He, M. Vechev, Large Language Models for Code: Security Hardening and Adversarial Testing, in: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 1865–1879. URL: https://doi.org/10.1145/3576915.3623175. doi:10.1145/3576915.3623175.

[11] S. Kalra, S. Goel, M. Dhawan, S. Sharma, Zeus: analyzing safety of smart contracts, in: Ndss, 2018, pp. 1–12.

[12] E. Zamani, Y. He, M. Phillips, On the security risks of the blockchain, Journal of Computer Information Systems 60 (2020) 495–506.

[13] N. Szabo, Formalizing and securing relationships on public networks, First monday (1997).

[14] P. Zheng, Z. Jiang, J. Wu, Z. Zheng, Blockchain-Based Decentralized Application: A Survey, IEEE Open Journal of the Computer Society 4 (2023) 121–133. doi:10.1109/OJCS.2023.3251854.

[15] OWASP Foundation, OWASP Smart Contract Top 10, https://owasp.org/www-project-smart-contract-top-10/, 2025. Accessed: 2025-07.

[16] T. Jiao, Z. Xu, M. Qi, S. Wen, Y. Xiang, G. Nan, A Survey of Ethereum Smart Contract Security: Attacks and Detection, Distrib. Ledger Technol. (2024). URL: https://doi.org/10.1145/3643895. doi:10.1145/3643895.

[17] V. Arceri, S. M. Merenda, G. Dolcetti, L. Negrini, L. Olivieri, E. Zaffanella, Towards a sound construction of EVM bytecode control-flow graphs, in: Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2024, ACM, 2024, pp. 11–16. doi:10.1145/3678721.3686227.

[18] M. Mehar, C. Shier, A. Giambattista, E. Gong, G. Fletcher, R. Sanayhie, H. M. Kim, M. Laskowski, Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack, in: Journal of Cases on Information Technology 21, 2024. URL: arXiv:2401.14196.

[19] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, R. Hierons, Smart contracts vulnerabil-

ities: a call for blockchain software engineering?, in: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), 2018, pp. 19–25. doi:10.1109/IWBOSE.2018.8327567.

[20] C. Ferreira Torres, A. K. Iannillo, A. Gervais, R. State, The Eye of Horus: Spotting and Analyzing Attacks on Ethereum Smart Contracts, in: Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I, Springer-Verlag, Berlin, Heidelberg, 2021, p. 33–52. URL: https://doi.org/10.1007/978-3-662-64322-8_2. doi:10.1007/978-3-662-64322-8_2.

[21] S. Kim, S. Ryu, Analysis of Blockchain Smart Contracts: Techniques and Insights, in: 2020 IEEE Secure Development (SecDev), 2020, pp. 65–73. doi:10.1109/SecDev45635.2020.00026.

[22] C. Sendner, L. Petzi, J. Stang, A. Dmitrienko, Large-Scale Study of Vulnerability Scanners for Ethereum Smart Contracts, in: 2024 IEEE Symposium on Security and Privacy (SP), IEEE Computer Society, Los Alamitos, CA, USA, 2024, pp. 220–220. URL: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00230. doi:10.1109/SP54263.2024.00230.

[23] J. Feist, G. Grieco, A. Groce, Slither: A Static Analysis Framework for Smart Contracts, 2019. URL: https://arxiv.org/abs/1908.09878.

[24] Crytic, Slither: The Smart Contract Static Analyzer, https://crytic.github.io/slither/slither.html, 2019. Accessed: 2025-07.

[25] N. Sharma, S. Sharma, A Survey of Mythril, A Smart Contract Security Analysis Tool for EVM Bytecode, Indian Journal of Natural Sciences 13 (2022) 51003–51010.

[26] Consensys, Mythril GitHub Repository, 2025. URL: https://github.com/ConsenSys/mythril, accessed: 2025-07-26.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Advances in neural information processing systems 30 (2017).

[28] 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 33 (2020) 1877–1901. URL: https://doi.org/10.5555/3495724.3495883. doi:10.5555/3495724.3495883.

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

[30] B. Acharya, M. Saad, A. E. Cinà, L. Schonherr, H. D. Nguyen, A. Oest, P. Vadrevu, 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. URL: https://doi.org/10.1109/SP54263.2024.00156. doi:10.1109/SP54263.2024.00156.

[31] B. Acharya, D. Lazzaro, A. E. Cinà, 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. URL: https://doi.org/10.1145/3696410.3714634. doi:10.1145/3696410.3714634.

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

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

[34] Y. Wei, C. S. Xia, 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. URL: https://doi.org/10.1145/3611643.3616271. doi:10.1145/3611643.3616271.

[35] D. Beste, G. Menguy, H. Hajipour, M. Fritz, A. E. Cinà, S. Bardin, T. Holz, T. Eisenhofer, 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. URL: https://doi.org/10.1007/978-3-031-97620-9_15. doi:10.1007/978-3-031-97620-9_15.

[36] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, R. Karri, Asleep at the keyboard? assessing the security of github copilot's code contributions, in: 2022 IEEE Symposium on Security and Privacy

(SP), IEEE, 2022, pp. 754–768.
[37] R. Khoury, A. R. Avila, J. Brunelle, 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.
[38] 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 preprint arXiv:2305.06161 (2023).
[39] R. Karanjai, E. Li, L. Xu, W. Shi, Who is Smarter? An Empirical Study of AI-Based Smart Contract Creation, in: 2023 5th Conference on Blockchain Research and Applications for Innovative Networks and Services (BRAINS), 2023, pp. 1–8. doi:10.1109/BRAINS59668.2023.10316829.
[40] E. A. Napoli, V. Gatteschi, Evaluating ChatGPT for Smart Contracts Vulnerability Correction, in: 2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC), 2023, pp. 1828–1833. doi:10.1109/COMPSAC57700.2023.00283.
[41] F. Barbàra, E. A. Napoli, V. Gatteschi, C. Schifanella, Automatic smart contract generation through llms: When the stochastic parrot fails, in: 6th Distributed Ledger Technology Workshop, 2024.
[42] L. Olivieri, D. Beste, L. Negrini, L. Schönherr, A. E. Cinà, P. Ferrara, Code generation of smart contracts with llms: A case study on hyperledger fabric, in: 2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2025.
[43] L. Olivieri, L. Negrini, V. Arceri, P. Ferrara, A. Cortesi, Detection of read-write issues in hyperledger fabric smart contracts, in: Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing, 2025, pp. 329–337.
[44] L. Olivieri, L. Negrini, V. Arceri, B. Chachar, P. Ferrara, A. Cortesi, Detection of phantom reads in hyperledger fabric, IEEE Access 12 (2024) 80687–80697. doi:10.1109/ACCESS.2024.3410019.
[45] R. L. Jonas Oppenlaender, J. Silvennoinen, Prompting AI Art: An Investigation into the Creative Skill of Prompt Engineering, International Journal of Human–Computer Interaction 0 (2024) 1–23. URL: https://doi.org/10.1080/10447318.2024.2431761. doi:10.1080/10447318.2024.2431761. arXiv:https://doi.org/10.1080/10447318.2024.2431761.
[46] braindao, Solidity Dataset with Prompts, 2024. URL: https://huggingface.co/datasets/braindao/solidity-dataset.
[47] C. Osborne, J. Ding, H. R. Kirk, The AI community building the future? A quantitative analysis of development activity on Hugging Face Hub, Journal of Computational Social Science 7 (2024) 2067–2105. URL: http://dx.doi.org/10.1007/s42001-024-00300-8. doi:10.1007/s42001-024-00300-8.
[48] I. HuggingFace, Huggin Face, 2016. URL: https://huggingface.co/.
[49] Meta, The LLaMA 3 Herd of Models, 2024. URL: https://arxiv.org/abs/2407.21783. arXiv:2407.21783.
[50] seyyedaliayati, Solidity Dataset, 2023. URL: https://huggingface.co/datasets/seyyedaliayati/solidity-dataset.
[51] SWCRegistry, Smart Contract Weakness Classification, https://swcregistry.io/, 2025. Accessed: 2025-07-26.