IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 20 September 2024, accepted 21 October 2024, date of publication 25 October 2024, date of current version 15 November 2024.

*Digital Object Identifier 10.1109/ACCESS.2024.3486365*

RESEARCH ARTICLE

# Assessing GitHub Copilot in Solidity Development: Capabilities, Testing, and Bug Fixing

**GAVINA BARALLA, GIACOMO IBBA, AND ROBERTO TONELLI**
Department of Mathematics and Computer Science, University of Cagliari, 09124 Cagliari, Italy

Corresponding authors: Gavina Baralla (gavina.baralla@unica.it) and Giacomo Ibba (giacomof.ibba@unica.it)

This work was supported in part by the Programma Regionale di Sviluppo (2020–2024)—Regione Autonoma della Sardegna (RAS) Strategia 2—Identità Economica Progetto 2.1—Ricerca e Innovazione Technologica Progetto ''BandzAI+''—Bando Aiuti per Progetti di Ricerca e Sviluppo— Information and Communication Technology Sector (Settore ICT), Codice Unico di Progetto (CUP), under Grant F23C23000230008 and Grant G27H23000270002; in part by U.S. Department of Commerce under Grant BS123456; and in part by the ''IMASS CHAIN—Infrastructure Management Support System Chain'' co-funded under the National Military Research Plan 2020 under Grant CIG: 884399685F and Grant CUP: D84H22001380001.

**ABSTRACT** In the rapidly evolving landscape of blockchain technology, the development of reliable and secure smart contracts represents one of several crucial challenges. GitHub Copilot, an AI-powered code assistant, aims to enhance developer productivity by generating code snippets, facilitating testing, and assisting in program repair. This research examines Copilot's proficiency in generating functional and secure smart contracts, including token creation adhering to standards such as ERC20, ERC721, and ERC1155 with various optional features. Additionally, the study assesses its effectiveness in common development tasks, including the implementation of widely employed libraries such as SafeMath. Through controlled experiments, the accuracy, efficiency, and security of the code generated by Copilot are evaluated. This evaluation identifies both its strengths in expediting the development process and its limitations in managing complex blockchain-specific logic and security considerations. The findings contribute to an expanded understanding of the role of AI-assisted programming in blockchain development, offering insights into how developers can best leverage such tools in creating and testing smart contracts. This research aims to guide both practitioners and researchers in the blockchain domain, advancing the discussion on integrating AI into software development workflows in the context of Solidity and smart contract development, underscoring the need for further research to address the challenges and opportunities presented by AI in blockchain technology.

**INDEX TERMS** GitHub copilot, solidity, smart contract, AI, bug fixing, vulnerability.

## I. INTRODUCTION

The emergence of blockchain technology has ushered in a new era characterised by the revolutionary impact of decentralised applications (dApps) [1] on sectors where scalability, trustworthiness, and privacy are crucial. This transformation is fundamentally altering how transactions and data are managed across various industries [2]. The blockchain's features are the ability to facilitate transparent, secure, and decentralised transactions. These features boosted the dApps development, revolutionising the market of traditional applications where control rests within a trusted authority. The core components of these applications are smart contracts (SCs) [3], which are self-executing contractual states stored on the blockchain that automate and enforce the terms of an agreement. In the Ethereum SC development context, the programming languages, Solidity[1] and Vyper,[2] serve as prominent programming languages for dApps development. However, Solidity stands out as the predominant choice

The associate editor coordinating the review of this manuscript and approving it for publication was Barbara Masini.

[1]https://soliditylang.org/
[2]https://docs.vyperlang.org/en/stable/

among developers due to its extensive adoption, established support, and recognised utility in this domain, making it a paramount tool for SC development. Additionally, the development and evolution of the Solidity programming language have been closely followed by an active developer community, reflecting the need for efficient and secure smart contract development practices [4]. Nonetheless, the complexity of smart contract coding, combined with the immutable and public nature of blockchain, demands high standards of reliability and security in development practices [5]. This pursuit demands substantial expertise, dedicated effort, and significant time investment from developers to implement robust development practices [6] that ensure the integrity and security of blockchain-based applications.

The emergence of artificial intelligence (AI)-assisted coding tools presents a novel approach to address these challenges. The landscape of AI-assisted development tools is rich and diverse, offering a range of solutions tailored to different programming needs. Tools like GPT-4, Claude AI, Google Bard, and Gemini AI leverage deep learning algorithms to provide code completions and suggestions, enhancing developer productivity across various programming languages, including Solidity for smart contract development. Their continuous evolution reflects the growing influence of AI in software development, promising to reshape the landscape of coding with intelligent, context-aware assistance.

GitHub Copilot, developed by GitHub in collaboration with OpenAI, is at the forefront of this evolution. As an AI-powered code assistant, Copilot aims to streamline the software development process. Copilot has been trained on a vast list of programming languages and a multitude of code repositories, but it is not specifically created for Solidity languages. However, this exposure to a wide range of coding patterns and scenarios equips Copilot with a nuanced understanding of smart contract development, enabling it to provide contextually relevant code suggestions and insights. Despite this, evaluating Copilot's utility in Solidity is crucial. This paper seeks to explore Copilot's capabilities in four key areas of Solidity smart contract development: generating code from scratch, aiding developers in smart contract implementation, detecting and fixing bugs, and generating unit tests to assess smart contract functionalities.

### A. MOTIVATIONS

Developing robust, secure, and trustworthy dApps is non-trivial [5], [7], and could require significant expertise [8], which necessitates a prolonged period of dedicated practice and experience. These challenges underscore the need for advanced tools that can assist developers in creating error-free code that adheres to best practices in security and efficiency. The decision to focus on GitHub Copilot in this study is predicated on its advanced capabilities and its potential to significantly impact the efficiency and reliability of smart contract development. The application of GitHub Copilot in Solidity development represents an interesting intersection of AI and blockchain technology. While Copilot offers the promise of enhanced efficiency and a reduced error rate, the peculiarities of SC development—such as the need for gas optimisation, adherence to security protocols, and the execution of irreversible transactions—pose unique challenges. Hence, the motivation for this study arises from the pressing need to explore and evaluate the effectiveness of AI-assisted programming tools like GitHub Copilot in the specific context of blockchain and SC development, which requires particular dedication to security, robustness, and trustworthiness. Once deployed on the chain the code cannot be updated (or can hardly be updated by using specific patterns), making error-free and safe code crucial. With well-written code, gas, and thus money, can be stored during execution.

Smart contracts are often used to trade values (cryptocurrencies, tokens, and so on) and faulty code can be exploited by attackers to steal money or assets. Attackers can also aim at leveraging the possibility of a Denial of Service (DoS) attack, which can be critical when smart contracts can execute time-constrained code. Smart contract coding is still a new and evolving paradigm, with no decades of experience behind it like other programming paradigms, and many beginner SC developers may need help in writing and debugging code.

### B. CONTRIBUTIONS

This research aims to provide valuable insights into the capabilities and limitations of AI-assisted tools in software development, with a particular focus on Github Copilot's capability in creating blockchain-based solutions. The research addressed research questions are:

- **RQ1**: *How effectively does GitHub Copilot manage the complexities of Solidity programming?*
- **RQ2**: *How reliably does GitHub Copilot enhance the functionalities of contracts that provide basic features and structure?*
- **RQ3**: *What limitations does GitHub Copilot face in terms of security and logic complexity?*

To address the research questions, an evaluation of Copilot across four distinct scenarios was conducted to assess:

- Copilot's capabilities in **code generation** from scratch.
- Copilot's efficacy in providing **code assistance**.
- Copilot's proficiency in **vulnerability detection** and Automatic Program Reapir(APR).
- Copilot's proficiency to **generate unit tests**.

The outcomes of this study are intended to inform both academic research and practical development in the blockchain domain, guiding future innovations and implementations of AI technologies in software development workflows.

The remainder of the paper is organised as follows: Section II surveys prior research on AI-driven coding tools, focusing on GitHub Copilot's effectiveness in various programming languages and its application to Solidity smart contracts. Section III presents the research methodology of this paper, followed by Section IV, which analyses and

discusses the achieved results and Copilot's performance. Section V outlines the threats to validity and the potential limitations that may affect our work. Finally, Section VI discusses future work.

## II. RELATED WORK

This section provides an overview of prior research that leveraged AI and LLM for coding support and bug detection, both in a general programming context and with a specific emphasis on Ethereum smart contracts. The review covers empirical analyses of GitHub Copilot and other AI-based tools, comparing their performance across various programming languages. Finally, a comparison is made between existing literature and the unique contributions of our work, which addresses GitHub Copilot's application in Solidity smart contract development, filling a gap in the current body of research.

### A. AI-ASSISTED CODE GENERATION

The employment of artificial intelligence (AI) and large language models (LLM) to support and help developers in programming practices has seen widespread adoption [9]. Several studies have examined the capabilities and limitations of GitHub Copilot in various contexts. Nguyen and Nadi [10] explored the effectiveness of GitHub Copilot in providing code suggestions through an empirical study based on LeetCode programming questions. They tested different programming languages including Python, Java, JavaScript, and C, noting Copilot's ability to produce understandable solutions with low complexity. However, they identified limitations such as generating suboptimal code and using undefined helper methods.

Further examining Copilot's broader implications, Pearce et al. [11] evaluated GitHub Copilot's response to security scenarios across Python, C, and Verilog programming languages. They attributed many vulnerabilities to the open-source code used in Copilot's training, suggesting incorporating security tools during training and generation phases of tools like Copilot to enhance security responses.

Mastropaolo et al. [12] explored the robustness of DL-based code recommendation systems like Copilot using a dataset of 892 Java methods. They find that approximately 46% of the time, semantically equivalent descriptions often led to different code generation outcomes. Similarly, Yetistiren et al. [13] evaluated the code quality of GitHub Copilot, Amazon CodeWhisperer, and ChatGPT, tested for Python programming language. They observed that ChatGPT outperformed the other tools in generating correct code solutions. Authors also highlighted the impact of input quality, showing that clear problem descriptions are crucial for successful code generation.

Ciniselli et al. [14] conducted an empirical study on the use of RoBERTA for code completion tasks. Their work explored code completion at various granularity levels, including single tokens, entire statements, and code blocks. The study evaluated RoBERTa's performance on both Java and Android codebases, using raw and abstracted code representations. The achieved results show that RoBERTA represents a viable solution for code completion, with perfect predictions ranging from 7% when asking the model to guess entire blocks, up to 58%, reached in the simpler scenario of a few tokens masked from the same code statement

Fried et al. [15] introduced InCoder, a unified generative model for both program synthesis and code editing, mainly focused on the Python programming language. InCoder uses a causal masking objective during training, which allows it to infill arbitrary regions of code conditioned on both left and right contexts. Their results showed that InCoder's ability to condition on bidirectional context substantially improved performance on these tasks compared to left-to-right-only models, while still performing comparably on standard program synthesis benchmarks.

Wang et al. [16] introduced CodeT5, a unified pre-trained encoder-decoder model for code understanding and generation tasks. CodeT5 builds on the T5 architecture and incorporates two novel pre-training tasks: identifier-aware denoising and bimodal dual generation. The identifier-aware objective enables the model to better leverage the crucial token type information in code, while the bimodal dual generation task improves natural language and programming language alignment. The study demonstrates the effectiveness of incorporating code-specific knowledge into pre-training and the benefits of a unified encoder-decoder framework for both code understanding and generation tasks.

Our work enriches the current literature by focusing specifically on Solidity smart contract development using GitHub Copilot and comparing its performances with GPT-4 and GPT-3.5. Unlike previous studies that examined Copilot's performance in languages like Python, Java, and JavaScript, our research provides a comprehensive evaluation of Copilot's capabilities in generating Ethereum token standards (ERC20, ERC721, ERC1155), libraries, and other blockchain-specific contracts. We also assess Copilot's performance across varying levels of complexity, from simple to complex and real use cases smart contracts, which hasn't been extensively explored in previous works.

### B. AI ASSISTED PROGRAMMING

Mehmood et al. [17] analysed Copilot's test case generation capabilities, comparing them to manually written test cases. They emphasised the equivalent quality and effectiveness of AI-generated test cases but limited their study to Python and a small set of files.

Sobania et al. [18] compared GitHub Copilot and Genetic Programming (GP) for program synthesis, tested for Python languages, finding GP more suitable for problems requiring many input/output examples, while Copilot was preferable for problems defined by textual descriptions. In exploring productivity and code quality, studies have shown mixed results regarding GitHub Copilot's impact on developer

efficiency. While Copilot has been found to increase productivity by generating substantial amounts of code, it often produces lower-quality outputs that require significant debugging, highlighting the importance of balancing productivity with code quality [19]. Researchers have noted a strong correlation between Copilot's acceptance rate and perceived productivity, but caution that this metric alone does not fully capture the complexity of developer excuiperiences [20]. Additionally, while Copilot can enhance overall programming efficiency, it does not always reduce task completion time due to the need for additional debugging of AI-generated code [21].

Mozannar et al. [22] evaluate data drawn from interactions with GitHub Copilot, introducing a utility-theoretic framework to drive decisions about suggestions to display versus withhold. Using data from 535 programmers, they show that is possible to avoid displaying a significant fraction of suggestions that would have been rejected by developers. They also demonstrate the importance of incorporating the programmer's latent unobserved state in decisions about when to display suggestions, and showcase how using suggestion acceptance as a reward signal for guiding the display of suggestions can lead to suggestions of reduced quality.

Kostianis et al. [23] conducted a comprehensive review of AI-assisted programming tasks, focusing on the application of code embeddings and transformers. The authors analysed nine key programming tasks, and examined various approaches, such as Flow2Vec and FRET, a functional reinforced transformer with BERT. The study also highlighted the effectiveness of transformers in bug detection and correction, with models like CodeBERT demonstrating improved predictive accuracy across different software versions and projects.

Our work contributes to related literature by evaluating GitHub Copilot's performance in both code generation and implementation assistance specifically for Solidity smart contracts. Unlike previous studies that focused on general programming tasks or other languages, we provide insights into Copilot's ability to enhance smart contract development workflows. We also compare Copilot's code implementation assistant with its chat feature, offering a nuanced understanding of the tool's strengths and limitations in the context of blockchain development.

### C. AI APPLICATION IN SMART CONTRACT DEVELOPMENT AND SECURITY

Expanding on the application of AI in smart contract development, Karanjai et al. [24] investigated large language models like OpenAI's ChatGPT and Google's Palm2 in generating Solidity smart contract code. Their study identified challenges and limitations such as the non-deterministic nature of code outputs and the lack of reproducibility due to ongoing retraining of underlying models. They noted that templated prompts produced more compilable code, yet pointed out that both models introduced bugs at similar rates.

Dade et al. [25] also focused on optimising large language models for generating smart contract code they called MazzumaGPT. They examined hyperparameter effects and emphasised security and ethical implications, providing insights into enhancing developer productivity and code quality through AI-driven program synthesis in the blockchain domain.

Napoli et al. [26] propose a methodology that leverages the capabilities of LLMs to automate the generation of smart contracts, aiming to make accessible the development of smart contracts to inexperienced developers. They employ the CO-STAR methodology to optimise prompt creation, and they employ Slither to assess the safety of generated contracts. Their results show that the pipeline is able to produce 98.1% of compilable smart contracts, and the methodology produces valuable and consistent outputs.

Liu et al. [27] introduced PropertyGPT, an end-to-end LLM-driven formal verification pipeline for smart contracts. PropertyGPT leverages the in-context learning capabilities of large language models like GPT-4 to generate customised formal properties for unknown smart contract code. Their results show that PropertyGPT could generate high-quality properties, achieving an 80% recall compared to ground truth, and successfully detect many known vulnerabilities.

Leite et al. [28] employ ChatGPT to automatically infer formal specifications from component textual behavioural descriptions. Their proposed framework, called DbC-GPT, generates postcondition specifications for smart contract functions implemented in Solidity. They evaluated their framework employing Ethereum standards such as ERC20, ERC721, and ERC1155, and compared the precision of the generated specifications for several GPT contexts that consider information of these standards in isolation as well as their combination.

Our work contributes by providing a comprehensive analysis of GitHub Copilot's capabilities in multiple aspects of smart contract development, including code generation, implementation assistance, vulnerability detection, and unit test generation. While previous studies have explored the use of large language models for smart contract generation, our research offers an evaluation of the tool throughout the development lifecycle. We also assess Copilot's ability to detect and fix vulnerabilities in smart contracts, which is a critical aspect of blockchain security that hasn't been extensively explored in the context of AI-assisted development.

The Table 1 summarises the differences between existing works and our contribution.

## III. METHODOLOGY

To ensure a comprehensive understanding and to provide a structured approach for evaluating GitHub Copilot's capabilities in developing Solidity smart contracts, a methodology that encompasses a variety of testing criteria and contexts has been designed (see Figure 1). It is worth noting that this methodology is designed to be flexible and can be adapted to evaluate other AI models and Large Language Models (LLMs), such as Google Bard, or Anthropic's Claude,

**TABLE 1. Summary of related works and our contribution.**

| Ref. | Tool | Languages | Aspect Analysed |
|---|---|---|---|
| [10] | GitHub Copilot | Python, Java, JavaScript, C | Code suggestion generating code |
| [17] | GitHub Copilot | Python | Test case generation |
| [11] | GitHub Copilot | Python, C, Verilog | Security performance and vulnerabilities |
| [12] | GitHub Copilot | Java | Robustness of code recommendations |
| [14] | RoBERTA | Java, Android | Code generation |
| [15] | InCoder | Python | Code synthesis and code editing |
| [16] | CodeT5 | Multi-language | Code understanding and code generation |
| [13] | GitHub Copilot, Amazon CodeWhisperer, ChatGPT | Python | Code quality and correctness |
| [18] | GitHub Copilot, Genetic Programming | Python | Program synthesis |
| [19] , [21] | GitHub Copilot | Python | Impact on developers' productivity |
| [20] | GitHub Copilot | Python, Typescript, Java, Other | Impact on developers' productivity |
| [24] | ChatGPT, Palm2 | Solidity | Code generation quality, correctness, validity and security |
| [22] | GitHub Copilot | Multi-language | Code suggestion |
| [23] | Flow2Vec, FRET, CodeBERT | Multi-language | Programming tasks |
| [25] | MazzumaGPT | Solidity, Plutus | Optimisation, performance, functional correctness, security, ethics |
| [26] | GPT-4 | Solidity | Generating code from scratch |
| [27] | PropertyGPT | Solidity | Smart contract formal verification |
| [28] | DbC-GPT | Solidity | Smart contract formal verification |
| Our Work | GitHub Copilot | Solidity | Generating code from scratch Pair programming functionality, Unit test generation, Bug detecting and fixing |

by substituting the AI tool in the workflow. This adaptability allows for comparative studies and broader applicability of the research framework across various AI-assisted development tools.

This chapter outlines the methodology employed in the study, detailing the selection of AI-assisted development tools, the types of smart contracts (SCs) evaluated, and the parameters used to assess the generated code. In Figure 1 dashed arrows represent input and continuous arrows represent output. Yellow boxes encompass artifacts produced by external sources, and orange boxes encapsulate artifacts produced by GitHub Copilot.

GitHub Copilot was evaluated across four distinct aspects. First, its capability to generate code from scratch was assessed. Second, the features and performance of the code implementation assistant were examined. Third, its performance in vulnerability detection and automatic program repair (APR) was evaluated. Finally, its ability to generate unit tests for contracts generated from scratch was examined, comparing Copilot's functionality to GPT-4.

A) **Code Generation from Scratch:**
- *Scope:* Copilot's versatility and effectiveness were assessed by generating a wide spectrum of smart contracts.
- *Tested contracts*: The analysis focuses on three different categories: token standard contracts (ERC20,[3] ERC721 [29], ERC1155[4]), widely recognised Solidity libraries (e.g., SafeMath) and commonly used smart contracts from Remix,[5] (e.g. Storage, Ballot, and Owner).
- *Compilation and Deployment:* The implemented contracts were compiled and deployed using Truffle[6] to assess syntactic correctness, ensuring the code is free from syntactic errors.
- *Code quality evaluation:* The quality of the generated smart contracts was evaluated checking **Standard Adherence**, **Contextual Appropriateness** and **Completeness** with respect to the prompt, **Semantic Correctness** and **Vulnerability**.

B) **Code Implementation Assistant:**
- *Additional Functionalities:* The assistant's performance was assessed by asking it to implement additional functionalities related to the specific use cases of manually implemented SCs.
- *Performance Comparison:* The code assistant's performance was compared with Copilot's chat to determine which tool provides better, contextually relevant answers.
- *Compilation and Security Check:* It was verified that the resulting contracts could be successfully compiled through Remix and were scanned them with Slither [30] to check for vulnerability issues.

C) **Unit Tests:**
- *Scope*: Evaluating Copilot's functionality in generating unit tests.
- *Tested contract:* Copilot was asked to generate a unit test for the token contracts that Copilot created from scratch.
- *Evaluation:* The unit tests were compared with those generated by GPT-4.
- *Additional implication:* The generated unit tests allowed the verification of the semantic correctness of the token smart contracts.

D) **Bug Detection and Automatic Program Repair (APR):**
- *Dataset:* The SmartBugs [31] curated dataset[7] was employed, which includes 143 vulnerable contracts covering a significant range of SC vulnerability issues.
- *Vulnerabilities:* The preliminary analysis focused on three vulnerabilities: reentrancy, denial of service (DoS), and arithmetic exposures.
- *Evaluation Process:* Copilot was provided with the vulnerable code snippet (the entire contract) to detect

[3]https://docs.openzeppelin.com/contracts/4.x/erc20
[4]https://docs.openzeppelin.com/contracts/3.x/erc1155
[5]https://remix.ethereum.org/
[6]https://archive.trufflesuite.com/
[7]https://github.com/smartbugs/smartbugs-curated

and fix the exposures. If the initial attempt failed, two additional prompts were supplied: the second prompt identified the number and types of vulnerabilities, and the third prompt specified the exact lines of code (LOC) and related vulnerabilities.

- *Fix Evaluation:* Copilot's fixes were manually evaluated, considering an SC successfully patched if all reported vulnerabilities were addressed and the contract compiled after the fixing procedure. Additionally, the experiment was repeated employing GPT 3.5, providing the same prompts as Copilot's and the performance was compared to evaluate the most reliable tool.

The evaluation of code generation from scratch is presented in Section IV-A, of code implementation assistant in Section IV-B, of Copilot's performance in bug detection in Section IV-D and unit tests in Section IV-C.

## IV. ANALYSIS AND RESULTS

*Experimental Setup:* The experimental setup employed both an Asus ExpertBook equipped with a 12th Gen Intel(R) Core(TM) i7-1265U processor running at 1.80 GHz, featuring 40 GB of installed RAM (39.7 GB usable), and a MacBook Air equipped with an Apple M1 processor featuring 8 cores, 8 GB of RAM, and 256 GB of SSD storage. The Asus ExpertBook operates on a 64-bit system based on x64 architecture, running Windows 11 Pro, while the MacBook Air runs macOS Monterey version 12.6.6. Local installations of all the Solidity compiler versions were maintained to facilitate version switching as needed. To successfully run Truffle and Ganache, Node v20.12.2 and Web3.js v1.10.0 are also locally installed. Python 3.8.13 was also locally installed to run Slither successfully. Below are outlined the frameworks employed for this research and the related versions:

- *GitHub Copilot v1.188.0*
- *GitHub Copilot Chat v0.15.0*
- *Visual studio: version 1.89*
- *Truffle v5.11.5 (core: 5.11.5)*
- *Ganache v7.9.1*
- *Remix v0.48.0*

### A. CODE GENERATION FROM SCRATCH

This subsection examines GitHub Copilot's ability to autonomously generate smart contracts from scratch, a crucial capability for enhancing developer productivity and autonomy in coding.

1) **Categories of Smart contracts**
   - *Token Contracts:* These include digital asset contracts compliant with standards like ERC20, ERC721, and ERC1155.
   - *Libraries*: Essential utility libraries such as SafeMath, which are crucial for performing safe arithmetic operations without overflow errors.
   - *Common/Well-Known SCs*: Standard contracts like Simple Auction, Crowdfunding, Escrow, and Voting Contracts that are frequently utilised in various decentralised applications.
   - *Prompt*: The initial request for generating the contract from scratch was always made using the GitHub Copilot Assistant prompt. Only for subsequent iterations, the interaction with Copilot Chat was used to address errors and make necessary corrections.
2) **Evaluation Criteria.** The evaluation of the generated code covers several key aspects:
   - *SC (Smart Contract)*: Includes the name/typology of the contract.
   - *#It.*: Tracks the number of iterations or revisions the code underwent during the development and improvement process.
   - *Chat Interaction*: Evaluates the use of the Copilot chat feature to determine if interaction with the tool influenced the outcomes. Marked as Yes/No.
   - *Adherence to Standards*: Measures compliance with established programming standards and specific requirements, assessed as Yes, No, or Partial.
   - *Syntactic Correctness*: Checks whether the code is free from syntax errors and compiles correctly without warnings or errors. Marked as Yes/No. This is crucial for ensuring the technical integrity of the contract.
   - *Compl.*: Completeness evaluates whether the generated code fully meets the initial request given by the prompt. The levels of completeness are defined along with their associated percentage ranges as follows:
     - Low (0% to 60%: The code provides only some of the requested functionalities and lacks many critical elements specified in the initial request.
     - Medium (61% to 85%): The code covers most of the requested functionalities, though some may be incomplete or not optimised.
     - High(86% to 100%): The code fully satisfies all the requested functionalities with well-executed implementations, ready for production use without significant modifications.
   - *Vuln.*: Vulnerability analyses the code to identify and evaluate security risk management, including controls designed to prevent common vulnerabilities in smart contracts. Evaluated using Slither security tools.
3) **Analysis Overview.** For each type of smart contract, an evaluation matrix was populated covering all the aspects listed above. Detailed results for the three cases of smart contracts—tokens, libraries, and well-known SCs—are presented in the subsequent sections of this chapter.

   To minimise bias and ensure clarity in the assessment of GitHub Copilot's capabilities, a new chat session was created for each new smart contract. This approach helped isolate each generated code from any potential influence from previous sessions.

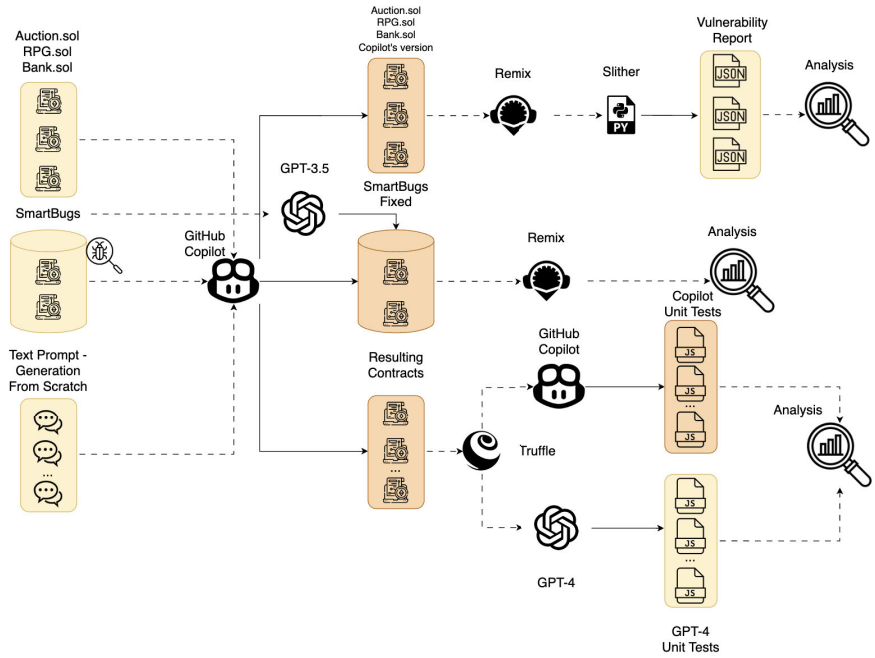


**FIGURE 1.** Proposed research methodology.

### 1) TOKEN

In the realm of Ethereum smart contracts, tokens represent a broad category of digital assets, each conforming to specific standards that define their behaviour and interactions on the blockchain. For this study, three widely recognised token standards were focused on due to their prevalent use and significance in the Ethereum ecosystem:

- ERC20: This is the standard interface for fungible tokens, meaning each token is exactly the same in type and value. ERC20 tokens are typically used for ICOs [32] (Initial Coin Offerings) and as a means to transact digital assets universally across various platforms.
- ERC721: Known for introducing non-fungible tokens (NFTs) on Ethereum, ERC721 tokens are unique and can represent ownership of specific assets (both digital and physical). Each token has a distinct value based on what it represents, making it ideal for collectibles and art.
- ERC1155: A newer standard that enhances the capabilities of ERC721 by allowing a single contract to manage both fungible and non-fungible tokens. This multi-token standard provides increased efficiency and flexibility in transaction processes.

These standards were chosen as test samples to evaluate GitHub Copilot's efficacy in generating code across a spectrum of complexities and functionalities inherent to blockchain technology. Given their foundational role in many decentralised applications, understanding Copilot's ability to handle such diverse and widely-used token types provides insights into its utility and limitations. To thoroughly assess Copilot's capabilities, six different prompts for each token type were constructed, ranging from level 0 (Extra simple) to level 5 (Ultra complex). The detailed prompt used for each of the 18 smart contracts generated is provided in Appendix A. This approach allowed the observation of Copilot's performance in generating smart contracts of varying complexity and the evaluation of its responsiveness and accuracy in adhering to the Ethereum standards. In total, 18 smart contracts were generated for these tokens.

A summary of the analyses conducted is presented in Table 2, which provides a detailed look at the outcomes for each generated contract across the specified complexity levels.

However, due to space constraints, some detailed aspects of the evaluation are not included within it.

*Chat Interaction*: the column related to the number of iterations (#It.) indirectly reflects the extent of chat interaction. A single iteration indicates that no interaction with the chat was necessary, implying that Copilot generated the code without errors on the first attempt. If the number of iterations exceeds one, it indicates that the chat was used to correct errors. To limit the bias, only the error output during compilation was provided to the chat, which then suggested corrections. This process was repeated until a compilable code was obtained. These interactions highlight the iterative nature of working with AI tools and their role in refining the code generation process by means an auto-learning processing.

*Syntactic Correctness*: all contracts eventually achieved syntactic correctness as indicated, after the number of iterations reported in the Table 2.

*Generated Code*: for a textual description and the complete code of each generated smart contract, please refer to the

**TABLE 2. Copilot's performance in generation token's smart contract.**

| SC | #It. | Prompt | Compl. | Sem. Corr. | Vuln. |
|---|---|---|---|---|---|
| ERC20_0.sol | 1 | Extra Simple | Hight | Yes | SAFE - Low |
| ERC20_1.sol | 2 | Simple | Hight | Yes | SAFE - Low |
| ERC20_2.sol | 5 | Detailed | Hight | Yes | SAFE - Low |
| ERC20_3.sol | 3 | Complex | Hight | Yes | SAFE - Low |
| ERC20_4.sol | 1 | Very Complex | Medium. | Yes | SAFE - Low |
| ERC20_5.sol | 2 | Ultra Complex | Low | No | SAFE - Low |
| ERC721_0.sol | 4 | Extra Simple | Hight | Partially | SAFE - Medium - UnusedReturn |
| ERC721_1.sol | 2 | Simple | Hight | Partially | SAFE - Medium - UnusedReturn |
| ERC721_2.sol | 1 | Detailed | Hight | Yes | SAFE - Medium - Unused Return |
| ERC721_3.sol | 1 | Complex | Hight | Partially | SAFE - Medium - Unused Return |
| ERC721_4.sol | 8 | Very Complex | Low | Partially | SAFE - Medium - Unused Return. Variables N.I. |
| ERC721_5.sol | 4 | Ultra Complex | Low | Yes | SAFE - Medium - Unused Return. Variables N.I. |
| ERC1155_0.sol | 1 | Extra Simple | Hight | Yes | SAFE - Medium - Unused Return. Variables N.I. |
| ERC1155_1.sol | 4 | Simple | Medium | Yes | SAFE - Medium - Unused Return. Variables N.I. |
| ERC1155_2.sol | 1 | Detailed | Medium | Yes | SAFE - Medium - Unused Return. Variables N.I. |
| ERC1155_3.sol | 1 | Complex | Medium | Partially | SAFE - Medium - Unused Return. Variables N.I. |
| ERC1155_4.sol | 1 | Very Complex | Medium | No | SAFE - Medium - Unused Return. Variables N.I. |
| ERC1155_5.sol | 5 | Ultra Complex | Low | Partially | SAFE - Medium - Unused Return. Variables N.I. |

replication package.[8] This package serves as a comprehensive resource for those interested in examining the specifics of the code produced during the study.

*Adherence to Standards*: all generated contracts were found to be compliant with established Ethereum standards, underscoring Copilot's capacity to align with industry norms and guidelines effectively.

*Semantic Correctness*: the semantic accuracy of the contracts was assessed through the creation of unit tests, conducted both by Copilot and GPT-4. The details of these tests and their outcomes are discussed in section IV-C, providing insights into how well the generated code aligns with its intended functionality and the specific requirements of smart contracts.

Based on data analysis from Table 2 we can summarise our results:

*Prompt Complexity and Code Completeness:* There is a clear correlation between the complexity of the prompt and the completeness of the generated code. Contracts generated from simpler prompts (Extra Simple and Simple) consistently achieved high completeness scores, whereas contracts from more complex prompts (Very Complex and Ultra Complex) often resulted in lower completeness, see Figure 2. To further refine the assessment, it was decided to also consider the number of iterations required to achieve a compilable contract within the completeness value. Each iteration beyond the first incurs a 2% penalty to the completeness score. This adjustment reflects the additional time and effort required to refine the contract to meet the specified requirements, highlighting the influence of the time spent on development on the final evaluation.

*Semantic Correctness:* Most contracts, irrespective of their complexity, maintained a high level of semantic correctness. This indicates that while the code may not fully meet the detailed requirements of complex prompts, its functions are generally correct and align with the basic specifications, see Figure 3.

*Security Analysis:* The vulnerability analysis conducted using Slither revealed that all contracts can be classified as "SAFE" suggesting that basic security measures are well handled by Copilot. However, several ERC721 and ERC1155 contracts were marked with "Medium" security concerns, observing issues like "Unused Return" and "Variables never Initialised." It is important to observe that these medium-level vulnerabilities primarily arise from code sections within the OpenZeppelin libraries[9] imported into the smart contracts, rather than direct implementations by Copilot. Furthermore, upon closer analysis of the Copilot-generated code, the variables identified as 'never initialised' were actually runtime-used variables where initialisation was not necessary. This highlights the importance of context in vulnerability assessments and the need for careful review to distinguish between genuine security risks and benign aspects of code structure.

*Gas consumption*: For the Copilot-generated contracts, a comparative evaluation of gas consumption was conducted, using OpenZeppelin contracts as a comparison. The primary aim was to understand how the implementation differences affect gas consumption, which in turn influences blockchain overhead and scalability. For the token contracts, the gas consumption data were evaluated for the main contract operations—such as minting, transferring, and burning tokens—and were collected under controlled tests using the Truffle Suite to deploy and interact with smart contracts on the Ganache CLI. Data are reported in a table comparing gas costs, an excerpt of which is presented in

[8]https://zenodo.org/records/11387353

[9]https://www.openzeppelin.com/

**TABLE 3. Excerpt from comparative analysis of gas consumption.**

| Contract Type | Operation | OpenZeppelin Gas | Copilot Gas | Gas Difference |
|---|---|---|---|---|
| ERC20_0 | Transfer | 52,137 | 52,339 | +202 |
| | Approve | 46,830 | 46,675 | -155 |
| | TransferFrom | 55,475 | 53,813 | -1,662 |
| ERC721_3 | Mint | 69,469 | 100,535 | +31,066 |
| | Transfer | 60,961 | 61,028 | +67 |
| | Burn | 32,123 | 38,071 | +5,948 |
| ERC1155_5 | Burn | 31,533 | 58,030 | +26,497 |
| | Mint | 50,055 | 72,530 | +22,475 |
| | Burn | 31,533 | 37,220 | +5,687 |

Table 3. This approach allowed for the isolation of contract functionality and measurement of gas used for predefined tasks without external influences that might be present on the main network. The gas cost was fixed to 20 gwei to maintain consistency across tests and provide a stable baseline for comparisons. For ERC20 contracts, minimal variations were observed in gas costs for standard operations such as transfer, approve, and transferFrom. Copilot's ERC20 implementations generally consumed slightly more gas in some instances, particularly for functionalities that included minting and burning. For instance, the ERC20_3 contract from Copilot consumed more gas during mint and burn operations compared to its OpenZeppelin counterpart. This indicates a potential area for optimisation in Copilot-generated contracts to reduce costs and blockchain load. In the ERC721 analysis, the enhanced features in some of Copilot's implementations, like advanced metadata handling and governance capabilities, led to higher gas consumption. This was evident in contracts like ERC721_1 and ERC721_3, where complex functionalities significantly increased gas costs. These findings highlight a trade-off between advanced features and their cost implications on transaction fees and blockchain performance. The ERC1155 contracts showed a notable increase in gas usage for batch operations in Copilot's implementations. For example, the ERC1155_3 contract used significantly more gas for batch minting and burning compared to OpenZeppelin. This reflects the added complexity and computational demand of Copilot's contracts, which could impact the scalability and efficiency of blockchain applications employing these tokens.

### 2) LIBRARIES

When developing SC and dApps, the employment of acknowledged and well-established libraries is crucial for bug prevention, enhancement of design patterns, optimisation, and compatibility. OpenZeppelin provides a wide range of libraries tailored to SC development, whose use cases range from mathematical operations, management of string and address types, and also contracts to prevent vulnerabilities such as reentrancies and arithmetic overflow and underflow. In this experimental session, a sample of libraries from OpenZeppelin contracts covering a plethora of use cases was selected, and Copilot was asked to generate the code from scratch for each library. The considered SC are:

- **ReentrancyGuard.sol**: ReentrancyGuard is an abstract contract that implements a lock-based mechanism to prevent the execution of reentrancy attacks by potential adversaries.
- **Address.sol**: The Address library encompasses a suite of functions designed to manage various operations related to the Address type in Solidity. These functions include verifying whether an address corresponds to a contract, and facilitating the transfer of ETH, among other related functionalities.
- **Math.sol**: The Math library delineates standard utilities for executing mathematical operations. In particular, it offers functions to ascertain the minimum, maximum, average, and perform rounding in division calculations.
- **SafeCast.sol**: SafeCast is a library that facilitates casting from uint256 to uint224, incorporating mechanisms to revert on overflow. This library is instrumental in mitigating numerous arithmetic issues like overflow and underflow.
- **SafeMath.sol**: SafeMath is among the most extensively employed libraries for the development of smart contracts necessitating the execution of mathematical operations susceptible to arithmetic overflow and underflow. However, from Solidity version 0.8.0 introduction, intrinsic mechanisms have been integrated into the compiler to directly prevent overflow and underflow.
- **Ownable.sol**: The contract offers a fundamental access control mechanism, facilitating the provision of exclusive access to a designated account.
- **Escrow.sol**: Escrow, operates as an escrow contract, safeguarding funds designated for collection by a user. It is designed to exclusively interact with the contract that instantiated it. The contract employing the escrow as a payment intermediary must act as its owner and provide public methods to redirect deposits and collections to the escrow.

In Copilot's input prompt, Copilot was asked to implement the contract by providing the guidelines written as comments by developers. This experimental session highlights several issues of Copilot's code generation assistance. It was observed that the *Address* contract is oversimplified since it misses several functions and does not check the successful execution of the *functionCall*, which introduces a potential threat to the security of the contract.

During the implementation of the *Math* contract, a misunderstanding of the requirements was observed. Copilot generated a version of SafeMath, which includes functionalities designed for the prevention of arithmetic issues such as overflow and underflow. However, this implementation did not align with the specified specifications, as a contract providing general arithmetic utilities was required. Consequently, an additional prompt was issued to clarify the request further.

Although Copilot implemented the matching functionalities for both *SafeCast* and *SafeMath*, some functions were missing. However, the problem was solved by asking

**TABLE 4. Copilot's performance in library and remix contract.**

| SC | #It. | Compl. | Vuln. |
|---|---|---|---|
| ReentrancyGuard.sol | 1 | Hight | SAFE - Low |
| Address.sol | 1 | Medium | SAFE - Low |
| Math.sol | 2 | Hight | SAFE - Low |
| SafeCast.sol | 1 | Hight | SAFE - Low |
| SafeMath.sol | 2 | Hight | SAFE - Low |
| Ownable.sol | 1 | Hight | SAFE - Low |
| Escrow | 1 | Hight | SAFE - Low |
| Owner.sol | 1 | Hight | SAFE - Low |
| Ballot.sol | 2 | Medium | SAFE - Low |
| Storage.sol | 1 | Hight | SAFE - Low |

Copilot's assistant to implement the missing functionalities, and consequentially, the contracts were considered successfully generated according to the specified criteria.

Contracts *Ownable* and *Escrow* implementation was assessed to test Copilot's assistant capabilities to deal with inheritance (the contract *Escrow* inherits from *Ownable*). The *Ownable* contract implementation required only one prompt to match OpenZeppelin's version. The *Escrow* contract exhibits slight differences in funds management tasks, even though the functionalities and code patterns match between the two versions. A significant difference with previous iterations consists in the complexity of the given prompt. Indeed, this is because the *Escrow* contract, which involves inheritance, is subject to specific constraints in fund management.

*Security Analysis:* Table 4 resumes the outcome of the analysis for both libraries and Remix contracts. Libraries like SafeMath, ReentrancyGuard, and Address play crucial roles in enhancing the security of SCs, making their correct implementation primarily important. Copilot, demonstrated a solid understanding of security patterns in libraries implementation. For ReentrancyGuard.sol, Copilot correctly implemented the lock mechanism to prevent reentrancy attacks, showing awareness of this critical security issue. Moreover, Copilot demonstrated domain knowledge and consistent implementation to prevent arithmetic issues such as overflows and underflows (e.g. SafeMath and SafeCast). However, some inconsistencies have been observed despite most libraries being implemented with high completeness scores. The Address library's implementation was oversimplified, not checking the successful execution of the *functionCall* function. Assuming the call function is used to transfer ETH, if the call fails, without proper checks funds may remain trapped in the contract, and resulting in a faulty contract. Moreover, without proper guards against reentrancy, an attacker may exploit the *functionCall* to completely drain the contract from its funds. Copilot, also misinterpreted the prompt for the *Math* library and generated a SafeMath-like contract instead of the requested general arithmetic utilities, pinpointing the importance of clear, specific prompts when dealing with security-critical components, and emphasising the necessity of prompt oriented to specific needs and developers' security requirements.

### 3) WELL KNOWN CONTRACTS

The experimentation was conducted by asking Copilot to implement contracts analogous to those already defined on Remix, which are:

- **Owner.sol**: This SC implementation provides the initialisation logic of the contract's owner and the logic to change the owner itself.
- **Storage.sol**: It is a trivial contract. It should save the value stored in a variable and return it (Remix's contract implements this logic for uint256 data).
- **Ballot.sol**: This SC implements a voting system and voting delegation.

Given the constrained number of functionalities and the trivial nature of the three considered smart contracts, prompts increasing in complexity are not provided. Thus, the effort was made to provide a prompt as trivial as possible. Copilot was asked to implement each contract with precise functionalities. The features required for the *Storage.sol* contract, are two functions allowing the storing and retrieving of a variable value respectively. With the exception of variable names and omitted comments, Copilot implemented a replication of the one available on Remix. It is noteworthy that a default implementation using pragma solidity ^0.8.0 is proposed, which might be influenced by the version of the compiler installed locally or in the Visual Studio plugin. According to the specified criteria, a single prompt was sufficient for Copilot to generate the contract accurately.

The *Owner* contract presents trivial features as the *Storage* SC. Nevertheless, it requires the implementation of the *isOwner* modifier, implementing logic triggering the execution of specific functionalities only by the contract's owner. Additionally, Copilot should implement an *OwnerChanged* event that registers the owner setting (thus, the event should be called whenever the contract's owner changes), and should also import the *console.sol* contract from the hardhat suite for logging purposes. In this experimentation, the contract generated by Copilot diverges from its Remix counterpart in only one aspect: the emission of the event. The Remix version emits the event both during the initialisation of the owner and when executing the function to change the owner. In contrast, Copilot's implementation emits the event solely during the owner change. Despite this difference, the smart contract maintains the same functionalities as the Remix version. According to the specified criteria, this is regarded as a successful generation.

The last SC is *Ballot*, which requires several functionalities. Indeed, it requires mappings, modifiers, events, and Structs, making it the most complex among the three contracts. Compared to the Remix implementation, several

constructs are missing. For instance, the *Proposal* structure, which is used to track the candidate's name and the number of votes received, is absent. Consequently, the history of candidates and their votes is also missing. Additionally, there is no variable to store the chair of the voting session, and the *Voter* struct lacks the variable required to record the accumulated weight from delegation. Several essential functions are also missing, such as the one that grants voting rights. The omission is related to the missing logic to manage the session chair, who is theoretically and logically responsible for granting voting rights. Furthermore, functions to return the winning proposal and the name of the winner are missing. This omission occurs because the contract generated by Copilot only manages a voting system without handling proposals. Hence, it was necessary to explicitly instruct Copilot to incorporate the management of proposals.

In this specific occurrence, a single prompt was not sufficient to implement the smart contract. Thus, Copilot was explicitly instructed to implement a session chair that provides authorisation for voting and the possibility to have multiple proposals for voters. The new prompt's outcome returns an SC structure with heightened complexity, bearing close resemblance to the original Remix version. The first difference in this instance is the omission of the *weight* variable to store the voting weight within the Voter structure results in the non-implementation of all operations associated with weight in the version proposed by Copilot. Additionally, Remix's original SC does not allow self-delegation, which is instead provided by Copilot.

*Security Analysis*: Considering simpler contracts like *Storage* and *Owner*, Copilot demonstrated consistent capabilities of implementing basic countermeasures, correctly implementing access control mechanisms and employing appropriate visibility specifiers. However, the tool's performance deteriorated when faced with the complexity of the *Ballot* contract, exposing gaps in its ability to consistently apply security patterns. The *Ballot* implementation lacked crucial structures for tracking candidates and votes, omitted essential access control for the voting chair, and failed to incorporate proper weight management for vote delegation. These are considered crucial system breaches, since illegal participants may be allowed to vote, and an attacker could impersonate the voting chair. The analysis also highlighted Copilot's inconsistent approach to transparency features, such as event emissions, underscoring the need for explicit security requirements in prompts.

**Answer to RQ1**: GitHub Copilot demonstrates proficiency in generating code for simpler smart contracts and standard token implementations (ERC20, ERC721, ERC1155). It performs well in adhering to basic security patterns and implementing standard functionalities. However, as contract complexity increases, Copilot's performance deteriorates in handling intricate logical constructs and addressing critical security considerations specific to blockchain development. This suggests that while Copilot is a valuable tool for basic Solidity programming tasks, it requires human oversight for more complex smart contract development.

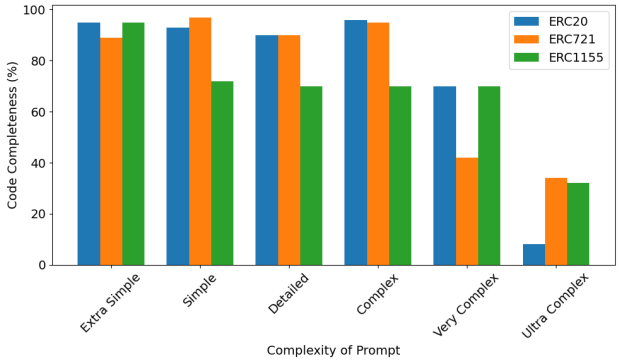


**FIGURE 2.** Prompt complexity vs code completeness by token type.

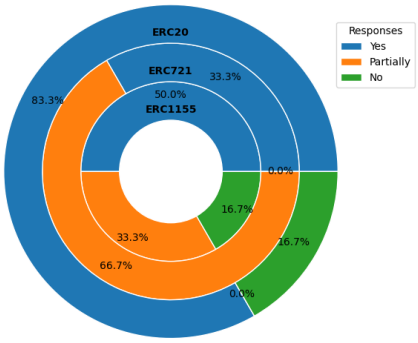


**FIGURE 3.** Semantic correctness by token type.

### B. CODE IMPLEMENTATION ASSISTANT

To assess the capabilities of the code implementation assistant, three smart contracts (SC) providing basic functionalities for popular smart contracts use cases were manually built, and the assistant was asked to implement additional functionalities for each contract. The contract implemented for this experimental session are:

- **Bank**: This contract implements a basic banking system. It features functionalities such as depositing funds, withdrawing funds, checking account balances, and incrementing earnings based on specific conditions. The contract includes several mappings to track account balances and the last time earnings were incremented for each account. The contract is owned by a specific address, granting the owner exclusive access to certain functions via a modifier.
- **Auction**: The contract implements a basic auction system designed to facilitate bidding and determine the highest bidder for a specific time period. It features functionalities to keep track of bids, the highest bidder, and the address of the beneficiary who will receive the highest bid amount. Moreover, it allows bidders to withdraw their bid, if they have been outbid.

**TABLE 5. Evaluation of Copilot's chat and code assistant on Bank, RPG, and SimpleAuction contracts.**

| Contract Name | Copilot | N. Functions | Comp. | Coher. | Exp. |
|---|---|---|---|---|---|
| Bank | Code assistant | 15 | YES | YES | NO |
| Bank | Chat | 7 | NO | YES | NO |
| RPG | Code assistant | 4 | NO | NO | NO |
| RPG | Chat | 8 | YES | YES | NO |
| SimpleAuction | Code assistant | 1 | NO | NO | NO |
| SimpleAuction | Chat | 6 | NO | YES | NO |

- **RPG**: This contract implements a role-playing game (RPG), allowing players to interact with the game by purchasing assets and managing player and item information, while ensuring secure token transfers and data storage using Solidity best practices.

In this experimental session, the additional functionalities implemented by the code assistant were evaluated comparing them with those implemented by Copilot's chat. The evaluation included the number of additional functionalities, syntactic correctness, and possible exposures introduced by both Copilot's chat and code implementation assistant. Table 5 summarises the evaluation outcomes of Copilot's chat and code implementation assistant. The column labeled *N. Functions* denotes the number of additional functions integrated, while the *Comp.* column indicates whether the generated contracts exhibit compilation errors. Additionally, the *Coher.* column assesses the degree of alignment between the implemented functions and the designated use cases of the examined contracts (e.g. a function allowing players to interact in the RPG contract). Finally, the *Exp.* column identifies whether the resultant contracts are exposed to any vulnerabilities, which are scanned through Slither. Observing Table 5, it is inferred that both Copilot's chat and code implementation assistant produce safe contracts in terms of vulnerability exposures (at least considering these basic contracts). In terms of Coherence, it was observed that the code assistant implemented functions strictly correlated to the specific use case in the *Bank* context. Indeed, the code assistant proposed 15 additional function related to tokens, and ETH transfer and withdrawal, which can be considered as additional features related to a banking system. Conversely, considering the *SimpleAuction* proposed only a fallback function, and considering the RPG proposed another fallback function and withdrawal and token transfer functions, which are not strictly related to the use cases of such contracts. Copilot's chat instead proposed additional functionalities for the three contracts. Within the *SimpleAuction* contract, the chat integrated:

- *Auction Start Time*: A start time feature in the auction contract, allowing for delayed auction initiation after deployment.
- *Minimum Bid Increment*: Implemented a minimum bid increment to prevent spam bids with very small increments, ensuring meaningful participation in the auction.
- *Owner Privileges*: Extended the contract to include an owner role with special privileges such as pausing, extending, or canceling the auction.
- *Multiple Rounds*: Enabled support for multiple bidding rounds within the auction contract, facilitating more complex auction scenarios.
- *Reserve Price*: Integrated a reserve price feature to set a minimum acceptable price for the auctioned item, ensuring auction success criteria.
- *Bid Visibility*: Concealed bid amounts until the end of the auction to maintain fairness and prevent bid influence.

The RPG contract has been enhanced with:

- *Player-to-Player Trading*: Implementation of asset trading between players.
- *Marketplace*: Creation of a platform for buying and selling items within the game.
- *Quests*: Introduction of tasks for players to earn rewards and advance in the game.
- *Leveling System*: Implemented character progression through experience accumulation.
- *Multiplayer Battles*: Enablement of player-versus-player combat interactions.
- *Item Rarity*: Addition of item rarity to confer unique bonuses based on rarity level.
- *Player Classes*: Development of distinct player classes with unique abilities and attributes.

The Bank contract was enhanced with the following features:

- *Interest System*: Development of a feature to accumulate interest over time based on user account balances.
- *Loan System*: Enablement of a borrowing system with loan tracking and repayment handling.
- *Transaction Fees*: Implementation of a system to deduct fees from each transaction.
- *Tiered Accounts*: Creation of different account types with varied benefits.
- *Pause Functionality*: Allow contract functions to be paused by the owner in emergencies.
- *Upgradeability*: Design the contract to support future feature additions without redeployment.
- *Audit Logs*: Implementation of transaction logging for auditing and accountability.

Upon comparing the implemented features, the chat's superiority over the code assistant was discerned. Copilot's chat improved the contracts with coherent features that substantially contributed to extending the functionalities of the three use cases. Furthermore, upon scanning the resulting contracts with Slither, none exhibited exposure to any Medium or High-impact vulnerability, ensuring their safety. Nonetheless, the experimental session revealed limitations for both Copilot's chat and code assistant, and a significant drawback of the chat emerges when adding new functionalities that require the code patterns implemented in previous interactions. For example, in the RPG contract, when implementing the Player class, Copilot's chat adds an enumeration to the Player struct, but the functions instantiating a new Player object are not updated with the new struct field, resulting in compilation errors.

Conversely, the code implementation assistant's suggestions for implementation are influenced by chat prompts.

When developing the Bank contract, it was observed that the code assistant proposed the implementation of a function called *buyAsset* that included elements from the RPG contract (which was the contract developed just before the Bank contract). This underscores the importance of employing a chat where interactions and requests are tailored to the contract currently under examination; otherwise, conflicts between the two tools may arise.

*Security Analysis*: The *SimpleAuction contract* is exposed to several vulnerabilities. A critical issue lies in the withdraw() function's exposure to reentrancy attacks, where an attacker could exploit the state update sequence to drain funds repeatedly. The contract's lack of access control mechanisms poses a significant threat, as it allows any address to execute sensitive functions like closeAuction(), disrupting the auction process. The presence of a receive() function without specific logic creates a pathway for ETH to be trapped in the contract, complicating fund management. Moreover, the absence of an auction start time feature limits deployment flexibility and may lead to premature bidding, and the contract's design overlooks the importance of a pause mechanism, leaving it vulnerable to continued exploitation in case of detected honeypots. Finally, the potential for an unbounded growth in the returnsPending mapping could result in gas limit errors during large-scale withdrawals, impacting the contract's long-term viability. Similarly to the *SimpleAuction*, the *RPG* contract lacks proper access control methods, and does not implement pausibility logic. However, the *RPG* contract, presents additional risks due to its more complex token economy and player interaction systems. The withdrawTokens function lacks proper access restrictions and balance tracking, allowing any address to drain the contract's tokens. The transferTokens function's flawed implementation, attempting to transfer tokens from the contract itself rather than user balances, could lead to unintended token movements. Furthermore, the contract's direct manipulation of player attributes in the buyAsset function without proper validation poses risks to game balance and integrity. The *Bank* contract shares several vulnerabilities with both the *SimpleAuction* and *RPG*, lacking proper access control, and implementing functions which logic may keep ETH trapped within the contract. However, this contract implements multiple redundant functions for transferring funds, significantly increasing the risk of potential attacks, introducing reentrancy vulnerabilities in its withdrawal functions. While the RPG contract had issues with token transfers, the *Bank* contract emphasises this problem with inconsistent and often unsecured methods for moving both Ether and tokens. Moreover, the *Bank* contract lacks event emission for critical state changes, the absence of withdrawal limits, and the improper implementation of token approvals. These issues, combined with the contract's overall lack of coherence and security best practices, make it significantly more vulnerable to exploitation than the previous two contracts.

### C. UNIT TEST GENERATION

In this research, the implementation of unit tests served two distinct purposes: firstly, to assess the quality of the unit tests themselves, which were generated by Copilot, and secondly, to evaluate the semantic correctness of smart contracts generated from scratch by GitHub Copilot (specifically tokens).

To provide a comprehensive evaluation of the quality of these unit tests, a comparative analysis was conducted with tests generated by another advanced AI tool, GPT-4. This comparison highlights the capabilities and limitations of Copilot in creating robust and effective test scenarios, offering insights into the potential enhancements needed for AI-generated tests to meet high standards of software testing.

#### 1) TEST GENERATION

Both GitHub Copilot and GPT-4 were employed to independently generate unit tests for each token smart contract. Copilot was integrated directly into the development environment, described in III, to provide initial test scripts, whereas GPT-4 was utilised via a web application. In total, 34 unit tests were performed, 17 from Copilot and 17 from GPT-4 (one for each smart contract token, the ERC20_5 contract was excluded because Copilot only provided the function headers and not its implementation). To ensure consistency, the same query, slightly modified, was used for both tools, as detailed in Appendix B.

#### 2) TEST CRITERIA

To evaluate the unit test code, an evaluation matrix composed of five different criteria was used, which are explained in detail below.

Evaluation criteria:

- *Functional Coverage* (Partial/Complete)). Measures the extent to which the generated unit tests cover the functionalities of the smart contract. ''Complete'' coverage means that tests are provided for all public and critical functions of the smart contract, while ''Partial'' coverage indicates that some functions may not be adequately tested or are omitted. This criterion ensures all functionalities are verified and the smart contract operates as intended under various scenarios.
- *Initialisation Verification* (Partial/Complete). It evaluates whether the unit tests check the correct initialisation of the smart contract's parameters such as names, symbols, decimals, and initial state settings. ''Complete'' indicates that all initialisation aspects are tested, while ''Partial'' suggests that some initial settings may not be verified. This criterion ensures the contract starts in a valid state, which is fundamental for its correct operation.
- *Error Handling* (Yes/No). It determines if the unit tests include scenarios that should lead to failures, such as insufficient balances or unauthorised actions. ''Yes'' indicates that error handling is tested, and

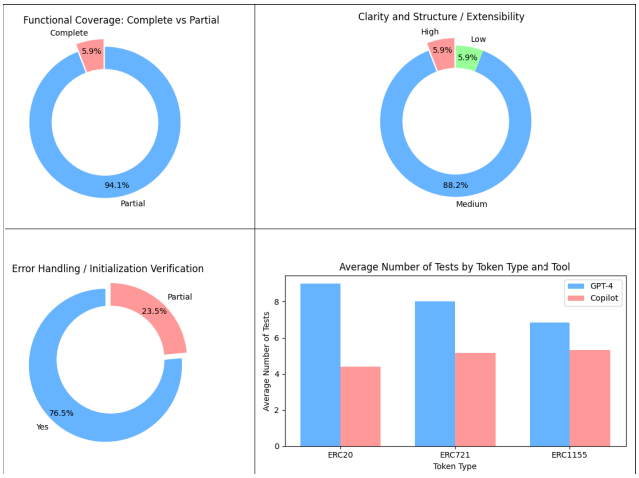


**FIGURE 4.** Copilot performance in unit test.

"No" means these scenarios are not covered. This tests the contract's robustness and security by ensuring it can gracefully handle and report errors, preventing unwanted behaviors.

- *Clarity and Structure* (Low/Medium/Hight). It assesses the readability and organisational structure of the test code. At the "Low" level, unit tests suffer from unclear purposes and functionalities due to insufficient documentation and disorganisation. Naming conventions are inconsistent and code structures are unnecessarily complex, complicating understanding and maintenance. The "Medium" level shows improvement with some commentary explaining logic, somewhat organised tests, and generally consistent naming, although some complexities remain in the code structure. At the "High" level, tests are exemplary: well-documented, logically organised, clearly named, and structured simply and elegantly, using best practices for easy maintenance and future modifications, facilitating efficient development processes.
- *Extensibility* (Low/Medium/Hight). It evaluates the ease with which new tests can be added or existing tests can be modified within the suite, without necessitating significant rewrites. "Low" value indicates it is hard to add or modify tests without affecting others; tests may be highly dependent on specific contract states or setups that are not reset between tests. "Medium" indicates some modularity and independence in tests, making modifications and additions somewhat straightforward, but improvements are possible. "High" value means tests are well-isolated with *beforeEach* setups, use reusable components, and are structured to easily accommodate changes and additions.

To conduct the analysis, an evaluation matrix was manually compiled for each test suite provided by Copilot and GPT-4. As an example, the compiled table for the smart contract ERC721_3 is presented, see Table 6.

From the comprehensive analysis, several insights can be deduced about Copilot's performance in generating unit tests for smart contracts, see also Figure 4:

- Coverage Gaps: Copilot often achieves only partial functional coverage in unit tests, which implies that it might not fully test all functionalities of the smart contracts. This partial coverage could leave certain aspects of the contract untested, exposing the contracts to risks and bugs that could have been identified through more thorough testing.
- Consistency in Initialisation Verification: Copilot generally performs well in terms of initialisation verification, often achieving complete coverage. This indicates that it is reliable in verifying that the smart contracts are correctly initialised with the expected initial parameters, which is critical for the contract's operation.
- Error Handling: There are indications that Copilot's error handling capabilities are sometimes incomplete. While it typically tests for some error scenarios, it may not be as thorough as GPT-4, potentially missing critical tests that ensure the contract behaves correctly under error conditions or when encountering unexpected inputs.
- Clarity and Structure: The clarity and structure of unit tests generated by Copilot are usually rated as medium. This suggests that while the test codes are functional, they may lack optimal organisation and documentation, making them harder to maintain and understand compared to the higher clarity observed in tests generated by GPT-4.
- Limited Extensibility: The tests generated by Copilot often exhibit medium extensibility. This indicates that while some tests can be modified or extended, doing so may not be straightforward. The tests might be somewhat rigid or tightly coupled to specific scenarios, which could make adapting the tests to changes in the contract more complex than necessary.
- Overall Effectiveness: While Copilot does create functional tests that can be used to assess some aspects of smart contracts, its effectiveness is somewhat limited by the issues noted above. Its performance in generating unit tests is adequate for basic functionality checks but might not meet the rigorous standards required for comprehensive and critical software testing environments.

These insights point to areas where Copilot could be improved, such as enhancing the comprehensiveness of its test coverage, improving the clarity and structure of the test code, and increasing the extensibility of its tests. These improvements could make Copilot more comparable to more sophisticated tools like GPT-4, which currently appears to offer more robust and thorough testing capabilities.

**Answer to RQ2**: GitHub Copilot demonstrates consistency in enhancing basic smart contract functionalities for standard token implementations and simpler use cases. It can effectively suggest additional features and implement common patterns, especially when guided by clear, specific

**TABLE 6. Evaluation matrix of unit tests performed to ERC721_3 sc.**

| ERC721_3.sol:Unit tests in Copilot were correctly configured after 6 iterations. | | |
|---|---|---|
| Criterion | GPT-4 | Copilot |
| Functional Coverage | Complete | Medium |
| Initialisation Verification | Complete | Complete |
| Error Handling | Yes | Yes |
| Clarity and Structure | High | Medium |
| Extensibility | High | Medium |
| Number of Tests | 14,<br>9 passing<br>5 failing | 7 passing |

prompts. However, the reliability decreases for more complex enhancements, and there's a risk of introducing inconsistencies or vulnerabilities when adding new functionalities to existing contracts. This suggests that while Copilot can be a valuable assistant for expanding basic contract features, developers should carefully review and test any enhancements for contracts with complex logic or high-value transactions.

### D. VULNERABILITIES DETECTION AND PROGRAM REPAIRING

Smart contracts (SCs) are exposed to a wide spectrum of vulnerability issues that may lead to irreversible consequences such as Ether leaking or data loss. For this very reason, developing and deploying safe programs in the Ethereum blockchain is crucial. However, developing vulnerability-free SCs requires significant expertise [33] and a deep knowledge of the Solidity programming language and its best practices. Several vulnerability detection tools and frameworks aid developers in the challenging task of developing secure smart contracts, such as Slither, Mythril [34], and Oyente [35]. Moreover, Artificial intelligence (AI) has demonstrated efficacy as a powerful tool for automating program repair and detecting vulnerabilities within software systems.

The aim of the experimental session is to assess the capability of GitHub's copilot to detect and automatically fix vulnerabilities within SCs code. The focus was placed on three different vulnerabilities:

- **Arithmetic overflow and underflow**: Arithmetic vulnerabilities persist as enduring challenges in programming, maintaining their relevance across various software development contexts. These exposures occur when a data value exceeds the storage capacity of the variable used to hold it, or in the case of underflow, when the number is too small to be represented. To mitigate this issue, developers can employ modifiers, Try-Catch-based controls, or, more simply and efficiently, the SafeMath library provided by OpenZeppelin. Versions of the Solidity pragma higher than 0.8.0 incorporate built-in methods to prevent these vulnerabilities.
- **Reentrancy**: In a reentrancy-attack scenario, an attacker intercepts an external call and exploits it to reenter multiple times into the pattern defined by this call. If this pattern involves operations such as money transfers or deposits, the attacker can deplete the funds of a contract. To mitigate this vulnerability, developers can employ check effect interaction patterns, boolean locks, and the ReentrancyGuard library provided by OpenZeppelin.
- **Denial of Service**: In the context of smart contracts, Denial of Service (DoS) refers to actions that lead the contract into an inoperable state, disabling its functionalities and features. Unfortunately, there is a wide range of issues that could lead a contract to an inoperable state, including gas exhaustion, improper use of selfdestruct/suicide calls, and reentrancy attacks. Consequently, the fix for these vulnerabilities varies depending on the specific issue leading to the DoS condition.

To assess Copilot's capabilities of vulnerability detection and automatic program repair (APR) the SmartBugs curated dataset[10] was employed, which encompasses several exposed SCs samples. The dataset provides several categories of vulnerabilities, including typically related Solidity vulnerabilities such as front-running and short addresses attacks. The smart contracts are organised into directories, each dedicated to SCs addressing a specific vulnerability. Within each file, a block comment is positioned at the beginning, providing details about the vulnerabilities addressed and the lines of code (LOC) that are affected. Additionally, comments are strategically placed within the exposed LOC to provide further context.

To evaluate Copilot's performance, the experimental setup was designed to present Copilot with three distinct prompts sequentially. The first prompt tasked Copilot with identifying and remedying vulnerabilities within a given smart contract. If Copilot failed to identify all vulnerabilities, a second prompt was issued specifying the number and types of vulnerabilities remaining. Lastly, if Copilot still did not detect all vulnerabilities, a third prompt provided the precise lines of code (LOC) that were vulnerable, along with the associated vulnerability types. As part of the preprocessing and code cleaning stage, all comments that could provide clues related to the vulnerabilities being identified were eliminated. This step is taken to ensure the fairness of Copilot's evaluation and fixing process by removing comments that could guide Copilot in correcting vulnerable patterns. Moreover, after several experimental prompts, it was observed that Copilot tends to employ features that might be incompatible with those required by the provided contract. Consequently, the prompt explicitly includes instructions on avoiding changing the pragma to a different version and refraining from using features that are incompatible with the pragma version specified by the contract.

The results are summarised in Table 7, which highlights Copilot's capability to fix most of the vulnerable snippets successfully. A SC is considered fixed if all the reported

[10]https://github.com/smartbugs/smartbugs-curated

**TABLE 7. Copilot's and ChatGPT's performance in the vulnerability detection session.**

| Contract name | Vuln | #V. | Comp. Cop. | Patc Cop. | Add. Vuln Cop. | #Pr Cop. | Comp. GPT | Patc GPT | Add. Vuln GPT | #Pr GPT |
|---|---|---|---|---|---|---|---|---|---|---|
| *tokensalechallenge.sol* | O&U | 1 | YES | YES | 1 | 2 | YES | YES | 1 | 1 |
| *token.sol* | O&U | 2 | YES | YES | 0 | 1 | YES | YES | 1 | 1 |
| *timelock.sol* | O&U | 1 | YES | YES | 1 | 1 | YES | YES | 1 | 1 |
| *integer_overflow_ single_tx.sol* | O&U | 6 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *overflow_simple_ add.sol* | O&U | 1 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *integer_overflow_multitx_ onefunc_feasible.sol* | O&U | 1 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *integer_overflow_multitx_ multifunc_feasible.sol* | O&U | 1 | YES | YES | 0 | 1 | YES | YES | 1 | 1 |
| *integer_overflow_mul.sol* | O&U | 1 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *integer_overflow_ minimal.sol* | O&U | 1 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *integer_overflow_ mapping_sym_1.sol* | O&U | 1 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *integer_overflow_ benign_1.sol* | O&U | 1 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *integer_overflow_1.sol* | O&U | 1 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *insecure_transfer.sol* | O&U | 1 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *BECToken.sol* | O&U | 1 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *auction.sol* | DoS | 1 | YES | YES | 1 | 2 | YES | YES | 2 | 1 |
| *dos_address.sol* | DoS | 1 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *dos_number.sol* | DoS | 1 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *dos_simple.sol* | DoS | 1 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *list_dos.sol* | DoS | 2 | NO | NO | 2 | 3 | YES | YES | 0 | 2 |
| *send_loop.sol* | DoS | 1 | YES | YES | 0 | 1 | YES | YES | 0 | 1 |
| *0x01f8c4e3fa3edeb29e514 cba738d87ce8c091d3f.sol* | Reentr | 1 | YES | YES | 1 | 3 | YES | YES | 3 | 1 |
| *0x4e73b32ed6c35f570686 b89848e5f39f20ecc106.sol* | Reentr | 1 | YES | YES | 2 | 1 | YES | YES | 3 | 1 |
| *0x7a8721a9d64c74da8994 24c1b52acbf58ddc9782.sol* | Reentr | 1 | YES | YES | 2 | 1 | YES | YES | 3 | 1 |
| *0x7b368c4e805c3870b6c4 9a3f1f49f69af8662cf3.sol* | Reentr | 1 | YES | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0x8c7777c45481dba41145 0c228cb692ac3d550344.sol* | Reentr | 1 | YES | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0x23a91059fdc9579a9fb d0edc5f2ea0bfdb70deb4.sol* | Reentr | 1 | YES | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0x93c32845fae42c83a70e 5f06214c8433665c2ab5.sol* | Reentr | 1 | YES | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0x96edbe868531bd23a6c0 5e9d0c424ea64fb1b78b.sol* | Reentr | 1 | NO | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0x561eac93c92360949ab1 f1403323e6db345cbf31.sol* | Reentr | 1 | YES | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0x627fa62ccbb1c1b04ffae cd72a53e37fc0e17839.sol* | Reentr | 1 | YES | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0x941d225236464a25eb1 8076df7da6a91d0f95e9e.sol* | Reentr | 1 | NO | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0x4320e6f8c05b27ab4707 cd1f6d5ce6f3e4b3a5a1.sol* | Reentr | 1 | NO | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0x7541b76cb60f4c60af330 c208b0623b7f54bf615.sol* | Reentr | 1 | NO | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0xaae1f51cf3339f18b6d3f3 bdc75a5facd744b0b8.sol* | Reentr | 1 | NO | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0xb5e1b1ee15c6fa0e48fce 100125569d430f1bd12.sol* | Reentr | 1 | NO | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0xb93430ce38ac4a6bb47f b1fc085ea669353fd89e.sol* | Reentr | 1 | NO | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0xbaf51e761510c1a11bf48 dd87c0307ac8a8c8a4f.sol* | Reentr | 1 | NO | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0xbe4041d55db380c5ae9d 4a9b9703f1ed4e7e3888.sol* | Reentr | 1 | NO | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0xcead721ef5b11f1a7b530 171aab69b16c5e66b6e.sol* | Reentr | 1 | YES | YES | 1 | 1 | YES | YES | 3 | 1 |
| *0xf015c35649c82f5467c9 c74b7f28ee67665aad68.sol* | Reentr | 1 | YES | YES | 1 | 1 | YES | YES | 3 | 1 |

vulnerabilities are detected and the exposed code patterns are successfully patched. The Table shows that Copilot was able to successfully fix all the samples exposed to arithmetic vulnerabilities, and most of the DoS instances. However, Copilot faced several issues in producing compilable contracts when analysing reentrancy-exposed SCs. All the resulting non-compilable SCs encompass a Log contract defining an *AddMessage* function used for logging functionalities. In each exposed sample, this function is called by another contract defined within the Solidity file, and after the fixing procedure, Copilot declares such function as internal. This leads to compilation errors since internal functions can be called by the defining contract. It has been observed that only 4 contracts of the 40 contracts selected sample required more than one prompt, which, in 3 occurrences was the first instance provided to Copilot for a specific category of exposure. This highlights Copilot's learning from previous data since all the other occurrences for the same category were fixed with only one required prompt.

Additionally, Copilot's performance was compared with ChatGPT's 3.5 using the same evaluation methodology. As shown in Table 7, ChatGPT successfully patched all the provided exposed samples, producing compilable SC and typically requiring only one prompt for patching, except in the case of one DoS-exposed contract. Furthermore, ChatGPT detected more additional vulnerabilities than those flagged within the SmartBugs dataset, demonstrating its superior reliability in this experimental session. These findings suggest that while Copilot shows promise, there is significant room for improvement, with ChatGPT emerging as the most reliable choice in the evaluation.

**Answer to RQ3**: Copilot shows limitations in consistently applying advanced security patterns and handling complex logic, especially in contracts with intricate functionalities. It struggles with implementing proper access controls, managing complex state transitions, and addressing blockchain-specific vulnerabilities like reentrancy. The tool's performance in vulnerability detection and automatic program repair is inconsistent, often requiring multiple prompts to identify and fix all issues. These limitations highlight the need for careful review and additional security measures when using Copilot-generated code in production environments.

## V. DISCUSSION AND LIMITATIONS

This section presents the outcomes of the research, delineating the strengths and limitations of Copilot while also addressing potential threats to the validity that could impact the findings.

### A. CODE FROM SCRATCH

Copilot shows promise in generating functional and secure smart contract code for common Ethereum token standards. It performs well with simpler tasks but has limitations for more complex requirements. This limitation leads to incomplete code and the need for more manual interventions. The results described from the tests were derived from using the Copilot Assistant prompt in the first iteration. From an initial analysis, it was observed that the same prompt in the chat provides a different, seemingly more performing contract. However, this issue was not further investigated for contracts generated from scratch. Additionally, it was observed that while GitHub Copilot Assistant provides consistent outputs for contracts, the chat tool exhibited some non-deterministic behavior. It is unclear whether this non-determinism is due to the AI nature of the tool or if some bias was unintentionally introduced, possibly through the interactions within the same chat. These questions are left open for future developments.

The generation of Library contracts from scratch revealed that Copilot's code assistant necessitates specific prompts. The experimental session demonstrated the tool's ability to handle complex use cases, such as inheritance, and produce functional and secure contracts in terms of vulnerability exposure. However, it also indicated that generic prompts might lead Copilot to misinterpret requirements, resulting in contracts with different functionalities, as observed in the *Math.sol* and *SafeMath.sol* test cases. This issue was further evidenced during the generation of Remix standard contracts, where Copilot's code assistant required more detailed prompts to provide a satisfactory implementation of the *Ballot.sol* contract. Thus, it is inferred that while Copilot's code assistant is useful for generating code from scratch, this process should be guided by detailed prompts. Additionally, Copilot's chat intervention may be necessary for more complex smart contract requirements.

### B. IMPLEMENTATION ASSISTANT

The comparison between Copilot's chat and code implementation assistant revealed that, as verified by Slither, both are capable of developing safe SC regarding vulnerability exposures. However, Copilot's chat demonstrated a higher degree of coherence by proposing functionalities closely aligned with the specified use cases, enhancing contracts' features. Despite these advancements, limitations arose with the chat's approach to implementing new functionalities, which sometimes failed to align with existing code patterns, causing compilation errors. Conversely, the code assistant's suggestions are more oriented on Solidity features rather than features related to use cases, and occasionally draw from unrelated prior interactions, highlighting the importance of tailored interactions to mitigate tool conflicts and enhance implementation accuracy.

### C. UNIT TEST

Copilot's functionalities in generating unit tests are useful for basic coverage and correct initialisation but show some gaps in terms of complete functional coverage, error handling, clarity of code structure, and test extensibility. While Copilot provides a good starting point for generating unit tests, improvements are needed to make it comparable to more sophisticated tools like GPT-4, which currently

offer more robust and comprehensive testing capabilities. Enhancing Copilot's test coverage, improving the clarity and organisation of the test code, and increasing the extensibility of its tests could significantly boost its effectiveness and reliability in critical software testing environments.

### D. VULNERABILITY DETECTION

When evaluating Copilot's performance in vulnerability detection and automatic program repair (APR), the findings underscore its ability to rectify most of the vulnerable code segments successfully. Indeed, Copilot effectively fixed all samples exposed to arithmetic vulnerabilities and most instances of DoS vulnerabilities, although it encountered challenges in producing compilable contracts when analysing reentrancy-exposed smart contracts (SCs).

Over a sample of 40 smart contracts, only a subset of 4 required more than one prompt, with three of these instances occurring as the first example for the representative category of vulnerabilities. This observation highlights Copilot's learning capability from previous data, as subsequent occurrences of the same exposure category were fixed with only one prompt. Nevertheless, comparing Copilot's and ChatGPT's performance, it is evident that OpenAI's framework is more reliable. Indeed, it was capable of fixing most instances with a single prompt, and additionally, it was capable of returning only compilable programs.

### E. COST EVALUATION

An evaluation of the communication costs for token contracts generated from scratch was driven by the need to provide a preliminary assessment of GitHub Copilot's impact in a well-defined and widely employed area of blockchain technology. The focus on token contracts for the gas consumption analysis was a strategic decision based on their standardised implementation and the availability of comparable data from established contracts such as those by OpenZeppelin. This preliminary evaluation was designed to assess the direct impact of GitHub Copilot's code generation on blockchain overhead and scalability for basic blockchain operations like minting, transferring, and burning tokens. The analysis revealed that while Copilot generally follows the functional benchmarks set by traditional standards, there are instances, especially in complex tasks like batch operations and advanced feature implementations, where gas consumption was higher than the OpenZeppelin equivalents. These findings underscore potential areas for optimisation in Copilot-generated contracts to enhance efficiency and reduce blockchain load. Such insights are crucial as they not only highlight the immediate cost implications of using AI-generated code but also set the groundwork for more extensive future studies aiming to refine AI tools for broader smart contract development. Moreover, considering the current limitations identified in Copilot's performance in more complex tasks such as security and comprehensive code quality, extending the cost analysis to library or assistant-generated contracts at this stage could obscure actionable insights. It was deemed essential to first understand the tool's capabilities in a controlled, standardised scenario before expanding the analysis to more complex and less predictable use cases. This phased approach allows for a clear attribution of findings to specific capabilities of the tool, providing a solid foundation for future explorations into more diverse contract types and their associated costs.

### F. SECURITY ANALYSIS

The security analysis of smart contracts generated by GitHub Copilot underscores the need for human expertise in guiding AI-assisted development. While Copilot exhibited consistency in implementing basic security patterns and libraries, such as SafeMath and ReentrancyGuard, its performance significantly varied across different contract complexities and use cases. For simpler contracts and standard libraries, Copilot adequately integrated fundamental security principles, implementing access control logic and arithmetic safeguards. However, as contract complexity increased, the tool's ability to consistently apply security patterns deteriorated. This gap was evident in the Ballot contract implementation, where crucial structures for vote tracking and access control were omitted, compromising the entire voting system's integrity. The analysis of more complex use cases, such as the *SimpleAuction*, *RPG*, and *Bank* contracts, exhibited a range of vulnerabilities from reentrancy risks to inadequate access controls and improper token handling. These issues highlight Copilot's limitations in comprehensively addressing the multifaceted security requirements of smart contracts. These findings demonstrate that while AI tools like Copilot can significantly accelerate the development process, they cannot replace the critical role of human expertise in ensuring smart contract security. Developers must approach Copilot-generated code (and AI) as a starting point rather than a finished product, subjecting it to rigorous review and enhancement. To effectively integrate Copilot-generated code into secure smart contract development practices, we propose the following guidelines:

- Explicit Security Requirements: Developers should include specific security considerations in their prompts to Copilot, guiding the AI towards implementing necessary safeguards.
- Comprehensive Review Process: All Copilot-generated code should undergo thorough manual review by experienced developers, focusing on potential security issues and adherence to best practices.
- Iterative Development: Adopt an iterative approach where Copilot-generated code is continuously refined and security measures are progressively integrated.
- Standardised Security Patterns: Develop and maintain a repository of secure code patterns and snippets that can be referenced in prompts to Copilot.

By adhering to these guidelines, developers can assess the efficiency of AI-assisted coding while mitigating its inherent limitations in understanding complex security contexts

Now, the discussion addresses the potential threats to validity that may affect the validity, the generalisability, and the applicability of the findings:

- **Evaluation of different smart contract standards**: When evaluating Copilot's capabilities in code generation from scratch, the focus was on the ERC token standard according to OpenZeppelin guidelines, which are widely recognised, and acknowledged by the Ethereum community. However, the analysis does not encompass contracts adhering to other dependencies standards (e.g. *pancakeswap*) posing a significant limitation since missing information about Copilot's capability of covering different standard contracts, also considering the complexity and challenges that such dependencies could introduce [36].
- **Comparison with other AI frameworks**: In the analysis, Copilot outcomes were compared only with GPT-4 in the evaluation of generating unit test. However, there is a lack of comparisons with other AI frameworks, such as *Claude AI*, *Google Bard* as well as with *GPT-4* regarding the other functionalities analysed. Comparing Copilot's with additional AI tools would contribute to highlighting the advantages and drawbacks of each technology, and pinpointing the most suitable AI assistant for SC development.
- **Unaware bias introduction** The analysis could have been influenced by the choice of the use cases and standards. For instance, asking Copilot's to implement additional functionalities for use cases different from those provided in this analysis may have led to different findings. Additionally, the order in which the SC and features have been implemented could have introduced a bias as well.
- **Limited sample in the vulnerability detection and APR evaluation**: Although Copilot has been proven capable of detecting and fixing a significant range of exposed smart contracts, the analysis is limited to only three vulnerabilities; namely, reentrancy, denial of service (DoS), and arithmetic vulnerabilities. Consequentially, the analysis should be expanded to assess additional vulnerability issues such as front running, time manipulation, short address attacks, and other exposures to improve the understanding of Copilot's capabilities, since being biased in fixing popular and widely acknowledged vulnerabilities is a concrete occurrence. Moreover, the select subset of contracts is limited and may not encompass all the possible code patterns that could lead to the considered exposures. Consequentially, enriching the dataset is crucial to improve the comprehension of SC vulnerability issues and the understanding of Copilot's capabilities in evaluating and detecting heterogeneous exposed code patterns.
- **Dependence on Training Data**: Copilot's performance is inherently tied to the quality and scope of the data it was trained on. If the training data does not comprehensively cover diverse smart contract patterns or includes outdated practices, Copilot's suggestions may be limited or suboptimal.
- **Scalability of Analysis**: The scalability of the evaluation process itself could be a concern. Conducting comprehensive evaluations across a broader range of smart contracts and different blockchain platforms would provide a more holistic view of Copilot's capabilities and limitations.
- **User Expertise Impact**: The level of expertise of the user interacting with Copilot can significantly influence the effectiveness of the AI tool. More experienced developers may be better at crafting effective prompts and identifying shortcomings in Copilot's output, leading to better overall outcomes.

## VI. CONCLUSION AND FUTURE WORKS

This research examined the efficacy of GitHub Copilot's chat and code implementation assistant, across various facets including code generation, smart contract (SC) development support, and vulnerability detection in the context of Solidity programming. Regarding RQ1, our analysis reveals that Copilot demonstrates proficiency in generating code for simpler smart contracts and standard token implementations, adhering to basic security patterns and fundamental functionalities. However, its performance declines as contract complexity increases, struggling with intricate logical constructs and blockchain-specific security considerations. Addressing RQ2, our study shows that Copilot reliably enhances basic smart contract functionalities, especially for standard token implementations and simpler use cases. It effectively suggests additional features and implements common patterns when guided by clear, specific prompts. However, its reliability decreases for more complex enhancements, risking the introduction of inconsistencies or vulnerabilities. In response to RQ3, we identified significant limitations in Copilot's ability to consistently apply advanced security patterns and handle complex logic. The tool's performance in vulnerability detection and automatic program repair is inconsistent, often requiring multiple prompts to identify and fix all issues. This underscores the need for careful review and additional security measures when using Copilot-generated code in production environments. These findings highlight Copilot's potential as a valuable assistant in smart contract development for basic tasks and standard implementations. However, they also emphasise the critical need for human oversight, especially in complex scenarios and security-critical aspects of smart contract development. Further research is needed to understand the non-deterministic behavior observed with GitHub Copilot Chat and to determine whether this variability is inherent to the AI tool or influenced by interaction biases.

As future research directions, the research aims to address the threats to validity that may affect the findings. Initially, the capability of Copilot in generating code from scratch will be assessed, introducing contracts adhering to different standards. The current analysis will be extended (including

only OpenZeppelin) to other contracts employed by the Ethereum community, such as *pancakeswap*, *0xprotocol*, *opensea*. This expansion will enrich our evaluation of Copilot's performance in generating such contracts and enhance our evaluation of Copilot's usefulness in assisting the development of decentralised applications (dApps), oriented to specific dependencies.

To provide an enhanced overview about the tools that may assist developer in writing Solidity SC, the analysis will be extended to include additional AI frameworks, such as *Claude AI*, *Google Bard*, and *Gemini AI*. The planned research consists in repeating the same improved analysis as those presented in this work for each AI, to assess which of the currently available tools provides the best performances, reliability, and safeness for Ethereum SC development.

Solving issues concerning manually introduced biases, is not a trivial task. It could depend on the text prompts' influence over the tool, or it could rely upon the complexity of smart contracts' source code provided as input. For this very reason, the scope of the analysis will be expanded to consider a wide spectrum of SC use cases, and explore whether there are design patterns for which Copilot issues with code generation and functionalities implementation. Moreover, the generation of text input introducing more complex prompt engineering methodologies will be considered, to explore how Copilot's answers change, and if this would produce better outcomes of those introduced in this preliminary analysis. The preliminary findings about cost consumption, indicate that while GitHub Copilot facilitates smart contract development, it does not inherently optimise gas consumption, a critical factor for blockchain efficiency. This initial analysis, primarily focusing on token contracts, underscores the need for more in-depth future studies aimed at enhancing gas efficiency across various contract types. Given these findings, future work should consider conducting a focused study on optimising gas consumption through specific prompts designed to challenge and enhance AI tools' efficiency. Such a study would entail a comprehensive comparison not only with Copilot but also with other AI coding assistants, exploring their effectiveness in reducing blockchain overhead in a range of contract types.

To improve the analysis on Copilot's proficiency in bug detection and automatic program repair (APR), the plan is to repeat the analysis on an extended range of vulnerabilities. Since the SmartBugs curated dataset is limited in terms of number of samples and variety of exposed patterns, the sample of vulnerable contracts looking for further examples and more complex vulnerability issues, stressing Copilot's performances, by also building specific test cases to assess how many false positives and false negatives Copilot detects.

The findings indicate that while AI-assisted programming tools like GitHub Copilot hold promise for enhancing software development, particularly in the blockchain domain, they are not yet capable of fully replacing human oversight, especially in contexts requiring high assurance of security and efficiency. As AI technology evolves, future iterations of tools like Copilot could see improvements that may address these gaps more effectively. As such, while Copilot can significantly accelerate the development process, relying on it to deliver secure and efficient code without human oversight remains premature. The tool's current iteration serves primarily as a first pass in generating code, which then requires rigorous testing and vetting for security flaws by experienced developers. By providing these insights, the intention is to contribute in the field of employment of AI tools for smart contract and dApps development. Furthermore, a replication package encompassing all the contracts, unit tests, and metadata generated by Copilot and GPT-4 is made available. Included is the subset selected from the SmartBugs dataset, including the contracts fixed by Copilot (including those that are non-compilable), encouraging developers and researchers to replicate the experiments or use the available metadata for their research.

The smart contracts and the unit tests developed by Copilot and GPT-4 are available on Zenodo[11] as provided by the two frameworks (non-compilable contracts are included as well) for study replication.

## APPENDIX A
## PROMPTS USED FOR EACH TOKEN TYPE

### ERC20 Token Generation Prompts

- Extra-simple: *Write an ERC20 smart contract.*
- Simple: *Write an ERC20 contract with standard transfer and balance functionalities.*
- Detailed: *Write an ERC20 contract using OpenZeppelin libraries to ensure security and standards compliance.*
- Complex: *Write an ERC20 contract that includes methods for minting, burning, and pausing, ensuring proper event handling and permissions management.*
- Very Complex: *Design an ERC20 smart contract that supports decentralised governance and can be upgraded via a proxy.*
- Ultra-complex: *Create an ERC20 contract that can interact with other blockchains and includes specific DeFi functionalities like staking, farming, and automatic swaps.*

### ERC-721 Token Generation Prompts

- Extra-simple: *Write an ERC721 smart contract.*
- Simple: *Write an ERC721 smart contract with standard transfer and balance functionalities for NFTs.*
- Detailed: *Write an ERC721 smart contract using OpenZeppelin libraries to ensure security and standards compliance.*
- Complex: *Write an ERC721 smart contract that includes methods for minting, burning, and pausing, ensuring proper event handling and permissions management.*
- Very Complex: *Design an ERC721 smart contract that supports decentralised governance and can be upgraded via a proxy.*

[11] https://zenodo.org/records/11387353

- Ultra-complex: *Create an ERC721 smart contract that can interact with other blockchains and includes specific functionalities for the digital art or gaming sectors.*

**ERC-1155 Token Generation Prompts**

- Extra-simple: *Write a basic ERC-1155 smart contract.*
- Simple: *Write an ERC-1155 smart contract with functionalities to mint, transfer, and check balances for multi-token types.*
- Detailed: *Write an ERC-1155 smart contract using OpenZeppelin libraries to ensure security, standards compliance, and support for batch operations.*
- Complex: *Write an ERC-1155 smart contract that includes advanced features such as batch minting, batch burning, and the ability to pause and resume token transfers.*
- Very Complex: *Design an ERC-1155 smart contract that supports decentralised governance, role-based access control, and can be upgraded via a proxy.*
- Ultra-complex: *Create an ERC-1155 smart contract that includes features tailored for specific industries.*

## APPENDIX B
## PROMPTS USED FOR LIBRARY AND REMIX CONTRACTS GENERATION

**ReentrancyGuard**: *Implement a contract module that helps prevent reentrant calls to a function.*

**Address**: *Implement a library called Address that defines a collection of functions related to the address type.*

**Math**:

- *Implement a library Math that implements standard math utilities missing in the Solidity language.*
- *The provided code implements the functionalities of the SafeMath library. You should implement missing math utilities, like returning maximum, minimum, and avg.*

**SafeCast**: *Implement a library called SafeCast that returns the downcasted uint224 from uint256, reverting on overflow (when the input is greater thanlargest uint224).*

**SafeMath**:

- *Implement a SafeMath library that wraps over Solidity's arithmetic operations.*
- *Implement the SafeMath library following OpenZeppelin's implementation. tryadd, trysub, trydiv, trymul and mod are missing.*

**Ownable**: *Implement an abstract contract Ownable that provides a basic access control mechanism, where there is an account (an owner) that can be granted exclusive access to specific functions. By default, the owner account will be the one that deploys the contract. This can later be changed with transferOwnership. This module is used through inheritance. It will make available the modifier onlyOwner', which can be applied to your functions to restrict their use to the owner. The contract inherits from Context.sol.*

**Escrow**: *Implement a base Escrow contract, which holds funds designated for a payee until they withdraw them. Intended usage: This contract (and derived escrow contracts) should be a standalone contract, that only interacts with the contract that instantiated it. That way, it is guaranteed that all Ether will be handled according to the 'Escrow' rules, and there is no need to check for payable functions or transfers in the inheritance tree. The contract that uses the escrow as its payment method should be its owner, and provide public methods redirecting to the escrow's deposit and withdrawal. This contract should inherit from Ownable.*

**Storage**: *Implement a Storage contract that stores and retrieves value in a variable.*

**Owner**: *Implement an Owner contract that sets and changes the owner of the contract. Import console.sol from Hardhat for logging.*

**Ballot**:

- *Implement a Ballot contract setting up a voting process along with vote delegation.*
- *You should implement additional functionalities. Implement a session chair that provides authorisation for voting and the possibility to have multiple proposals for voters.*

## APPENDIX C
## PROMPTS USED FOR UNIT TEST GENERATION

**QUERY Copilot**: *Write a suite of unit tests for smart contract in "filename", with the migration file in "filename", that ensure the contract is robust and functions as intended. The tests should cover*:

- *Initialisation and deployment correctness.*
- *Key functionalities typical for a token (transfer, balance management, etc.).*
- *Error handling to ensure the contract fails gracefully under improper conditions.*

*Please structure the tests using JavaScript for Truffle. Each test should be isolated to maintain independence. Focus on the following criteria for evaluating the quality of the tests*:

- *Functional Coverage: Ensure all major functions are tested.*
- *Initialisation Verification: Check that initial settings are correctly applied.*
- *Error Handling: Include tests for expected failures.*
- *Clarity and Structure: Tests should be clear and well-organised.*
- *Extensibility: Allow for easy addition and modification of tests.*

*This approach should provide a comprehensive and maintainable test suite that verifies the contract's functionality.*

**QUERY GPT-4**: *Write a suite of unit tests for smart contract with the code and the migration file below*

[code of sc and migration file…]

*The tests will ensure the contract is robust and functions as intended. The tests should cover*:

- *Initialisation and deployment correctness.*
- *Key functionalities typical for a token (transfer, balance management, etc.).*
- *Error handling to ensure the contract fails gracefully under improper conditions.*

*Please structure the tests using JavaScript for Truffle. Each test should be isolated to maintain independence. Focus on the following criteria for evaluating the quality of the tests*:

- *Functional Coverage: Ensure all major functions are tested.*
- *Initialisation Verification: Check that initial settings are correctly applied.*
- *Error Handling: Include tests for expected failures.*
- *Clarity and Structure: Tests should be clear and well-organised.*
- *Extensibility: Allow for easy addition and modification of tests.*

*This approach should provide a comprehensive and maintainable test suite that verifies the contract's functionality.*

## APPENDIX D
## PROMPTS USED FOR VULNERABILITY DETECTION AND PROGRAM FIXING

- First Prompt: *Detect and fix the vulnerabilities within this smart contract. Do not update the pragma version and don't use features incompatible with the provided pragma.*
- Second Prompt: *The provided smart contract is affected by additional vulnerabilities. The vulnerabilities are two reentrancies.*
- Third Prompt: *The lines of code (LOC) exposing this smart contract are: Line 60 exposed to denial of service and line 78 exposed to arithmetic overflow.*

## REFERENCES

[1] A. K. Tyagi, "Decentralized everything: Practical use of blockchain technology in future applications," in *Distributed Computing to Blockchain*. Amsterdam, The Netherlands: Elsevier, 2023, pp. 19–38.

[2] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," in *Proc. 9th Int. Conf. Comput., Commun. Netw. Technol. (ICCCNT)*, Jul. 2018, pp. 1–4.

[3] G. Ibba, S. Aufiero, S. Bartolucci, R. Neykova, M. Ortu, R. Tonelli, and G. Destefanis, "MindTheDApp: A toolchain for complex network-driven structural analysis of ethereum-based decentralized applications," *IEEE Access*, vol. 12, pp. 28382–28394, 2024.

[4] A. Pinna, S. Ibba, G. Baralla, R. Tonelli, and M. Marchesi, "A massive analysis of ethereum smart contracts empirical study and code metrics," *IEEE Access*, vol. 7, pp. 78194–78213, 2019.

[5] W. Zou, D. Lo, P. S. Kochhar, X. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2084–2106, Oct. 2021.

[6] L. Marchesi, M. Marchesi, and R. Tonelli, "ABCDE—Agile block chain DApp engineering," *Blockchain: Res. Appl.*, vol. 1, nos. 1–2, Dec. 2020, Art. no. 100002.

[7] C. Wu, J. Xiong, H. Xiong, Y. Zhao, and W. Yi, "A review on recent progress of smart contract in blockchain," *IEEE Access*, vol. 10, pp. 50839–50863, 2022.

[8] G. Ibba, S. Khullar, E. Tesfai, R. Neykova, S. Aufiero, M. Ortu, S. Bartolucci, and G. Destefanis, "A preliminary analysis of software metrics in decentralised applications," in *Proc. 5th ACM Int. Workshop Blockchain-Enabled Networked Sensor Syst.*, New York, NY, USA, Nov. 2023, pp. 27–33, doi: 10.1145/3628354.3629533.

[9] A. Beganovic, M. A. Jaber, and A. A. Almisreb, "Methods and applications of chatgpt in software development: A literature review," *Southeast Eur. J. Soft Comput.*, vol. 12, no. 1, pp. 8–12, 2023.

[10] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub Copilot's code suggestions," in *Proc. IEEE/ACM 19th Int. Conf. Mining Softw. Repositories (MSR)*, May 2022, pp. 1–5.

[11] 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 *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 754–768.

[12] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on GitHub copilot," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 2149–2160.

[13] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub copilot, Amazon CodeWhisperer, and ChatGPT," 2023, *arXiv:2304.10778*.

[14] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of BERT models for code completion," in *Proc. IEEE/ACM 18th Int. Conf. Mining Softw. Repositories (MSR)*, May 2021, pp. 108–119.

[15] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-T. Yih, L. Zettlemoyer, and M. Lewis, "InCoder: A generative model for code infilling and synthesis," 2022, *arXiv:2204.05999*.

[16] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder–decoder models for code understanding and generation," 2021, *arXiv:2109.00859*.

[17] S. Mehmood, U. I. Janjua, and A. Ahmed, "From manual to automatic: The evolution of test case generation methods and the role of GitHub copilot," in *Proc. Int. Conf. Frontiers Inf. Technol. (FIT)*, vol. 34, Dec. 2023, pp. 13–18.

[18] D. Sobania, M. Briesch, and F. Rothlauf, "Choose your programming copilot: A comparison of the program synthesis performance of GitHub copilot and genetic programming," in *Proc. Genetic Evol. Comput. Conf.*, Jul. 2022, pp. 1019–1027.

[19] S. Imai, "Is GitHub copilot a substitute for human pair-programming? An empirical study," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng., Companion*, May 2022, pp. 319–321.

[20] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proc. 6th ACM SIGPLAN Int. Symp. Mach. Program.*, Jun. 2022, pp. 21–29.

[21] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Proc. CHI Conf. Human Factors Comput. Syst. Extended Abstr.*, Apr. 2022, pp. 1–7.

[22] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "When to show a suggestion? Integrating human feedback in ai-assisted programming," in *Proc. AAAI Conf. Artif. Intell.*, 2024, vol. 38, no. 9, pp. 10137–10144.

[23] S. Kotsiantis, V. Verykios, and M. Tzagarakis, "AI-assisted programming tasks using code embeddings and transformers," *Electronics*, vol. 13, no. 4, p. 767, Feb. 2024.

[24] R. Karanjai, E. Li, L. Xu, and W. Shi, "Who is smarter? An empirical study of AI-based smart contract creation," in *Proc. 5th Conf. Blockchain Res. Appl. for Innov. Netw. Services (BRAINS)*, Oct. 2023, pp. 1–8.

[25] N. O. O. Dade, M. Lartey-Quaye, E. T.-K. Odonkor, and P. Ammah, "Optimizing large language models to expedite the development of smart contracts," 2023, *arXiv:2310.05178*.

[26] E. A. Napoli, F. Barbàra, V. Gatteschi, and C. Schifanella, "Leveraging large language models for automatic smart contract generation," in *Proc. IEEE 48th Annu. Comput., Softw., Appl. Conf. (COMPSAC)*, vol. 24, Jul. 2024, pp. 701–710.

[27] 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," 2024, *arXiv:2405.02580*.

[28] G. Leite, F. Arruda, P. Antonino, A. Sampaio, and A. Roscoe, "Extracting formal smart-contract specifications from natural language with LLMs," in *Proc. Int. Conf. Formal Aspects Compon. Softw.* Cham, Switzerland: Springer, 2024, pp. 109–126.

[29] S. Casale-Brunet, P. Ribeca, P. Doyle, and M. Mattavelli, "Networks of ethereum non-fungible tokens: A graph-based analysis of the ERC-721 ecosystem," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, Dec. 2021, pp. 188–195.

[30] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, May 2019, pp. 8–15.

[31] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "SmartBugs: A framework to analyze solidity smart contracts," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2020, pp. 1349–1352.

[32] G. Fenu, L. Marchesi, M. Marchesi, and R. Tonelli, "The ICO phenomenon and its relationships with ethereum smart contract environment," in *Proc. Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, Mar. 2018, pp. 26–32.
[33] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons, "Smart contracts vulnerabilities: A call for blockchain software engineering?" in *Proc. Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, Mar. 2018, pp. 19–25.
[34] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, Oct. 2020, pp. 530–541.
[35] S. Badruddoja, R. Dantu, Y. He, K. Upadhayay, and M. Thompson, "Making smart contracts smarter," in *Proc. IEEE Int. Conf. Blockchain Cryptocurrency (ICBC)*, May 2021, pp. 1–3.
[36] G. Ibba, G. Destefanis, R. Neykova, M. Ortu, S. Aufiero, and S. Bartolucci, "DAI: A dependencies analyzer and installer for solidity smart contracts," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng.-Companion (SANER-C)*, Mar. 2024, pp. 72–75.

**GAVINA BARALLA** received the Ph.D. degree in electronic and computer engineering from the Department of Electrical and Electronic Engineering (DIEE), University of Cagliari, in 2019, with a doctoral thesis titled "Enabling Technologies for the Development of Smart Cities," which proposed novel approaches for leveraging technology in urban development. She is currently an Assistant Professor Type A in computer science with the University of Cagliari. Her research interests include the intersection of blockchain technology and its applications across critical sectors, such as e-health, waste management, and agri-food supply chain optimization.

**GIACOMO IBBA** received the Ph.D. degree in mathematics and in computer science from the University of Cagliari, with a thesis titled "Toward Use Cases and Security: An In-Depth Ethereum Smart Contracts Study," which proposes an overview of Ethereum smart contracts, including security, vulnerabilities, use cases, categories, and analysis tools for decentralized applications, in 2024. He is currently a Postdoctoral Researcher with the Department of Mathematics and Computer Science, University of Cagliari. His main research interests include smart contracts security, blockchain oriented software-engineering, application of machine learning for smart contracts categorization, and exposure detection.

**ROBERTO TONELLI** is currently a Full Professor with the Mathematics and Computer Science Department, University of Cagliari, Italy. He is also works on blockchain technology and its applications and received the prize for the TOP-50 most influential papers on blockchain for 2018 from the Blockchain Connect Conference and invited to receive the prize and for giving a talk in San Francisco, in January 2019. He is also a Principal Organizer and the Chair of 14 International Workshops on Blockchain Oriented Software Engineering. He is also a Co-Organizer of the four editions of the "Summer School on Blockchain Technology and Distributed Ledgers," held in Pula, in June 2018, 2019, and 2022, and in September 2023, at DMI-Unica, organized by DMI with Unica. He is a member of International Association for Trusted Blockchain Applications (INATBA) Academy Board for Interoperating with EU. He is Representative of MISE for European Blockchain Partnership (EBP) for EU Commission on European Self Sovereign Identity Framework (ESSIF) use case for European Blockchain Service Infrastructure (EBSI) and a Representative of the University of Cagliari for Italian Blockchain Service Infrastructure (IBSI) in Italy.

Open Access funding provided by 'Università degli Studi di Cagliari' within the CRUI CARE Agreement