Secode: A Security Testing Framework for AI-Generated Code

Heying Chen*, Chang Zhou*, Yuqian Hong*
*University of Science and Technology of China

***Abstract*—1**
***Index Terms*—LLM, code secure,**

I. Introduction

Commercial large language models (LLMs) trained on large amounts of source code have been widely promoted as tools to help developers with general coding tasks, such as translating and interpreting code between programming languages, by predicting possible text completion given a few prompts, including comments, function names, and other code elements. Among the many tasks programmers do, we are interested in fixing security vulnerabilities; Developers may often run security tools such as fuzzers or static analyzers in an attempt to understand feedback, locate problems, and modify code to fix bugs.

In this article, we ask the question: Can large language models help us fix security vulnerabilities? Common large language models, such as OpenAI's GPT, are trained on open-source code in a myriad of languages that contain a lot of annotations and features, both buggy and non-buggy. This enhances the LLM's ability to complete the code in different ways, given some context, such as the designer's intent in the code comments.

Recent work has shown that completing code with GitHub Copilot may introduce security vulnerabilities, but Pearce et al.(Pearce et al. 2022 [6]) conclude that models can still "increase the productivity of software developers". Since it is possible to guide LLMs by adding hints to the prompts, do we develop large language models for source code security testing feedback?

Solidity is a programming language used for smart contract development, widely used in Ethereum and other blockchain platforms. Security issues are crucial for Solidity, as smart contracts are executed on the blockchain and their code cannot be changed. Any security vulnerability or weakness can lead to serious financial losses or contract attacks. Solidity security issues include but are not limited to reentry attacks, integer overflow, recursive calls, and permission issues. Therefore, developers must be very careful when writing Solidity smart contracts, adopting the best security practices to ensure the robustness of the contracts and the security of user assets.

Identifying security vulnerabilities in the C programming language is crucial due to its extensive usage, particularly in critical systems and open-source projects. C's low-level memory access poses challenges like buffer overflows and memory leaks, making it susceptible to exploitation. The reliability of critical systems, ranging from medical devices to industrial control systems, hinges on addressing these vulnerabilities. Additionally, legacy code written in C requires ongoing maintenance to meet modern security standards. Discovering and mitigating C language vulnerabilities not only secures individual systems but also contributes to industry-wide best practices, fostering a more robust and secure software ecosystem.

We wonder if LLMs – despite not being trained on specialized security fixes (and, in fact, on a lot of insecure code) – can still generate effective fixes for vulnerable code, particularly in Solidity and C. We seek answers to these research questions:

1) Can a paradigm be designed to correct in a multilingual environment and reveal security implications in the code generated by generative LLMs?
2) This solution not only focuses on solving the problem of language-specific generation detection, but also focuses on the research and practical application of generalized generative code security.

To answer these questions, we evaluate recent ChatGPT synthetic or hand-crafted buggy scenarios. Our contributions are as follows:

1) Secode, a standardized security detection scheme for intelligently generated code, was proposed. We have designed correctness and security solutions that can be used in multiple languages.
2) For the se curity detection of a certain language, we propose how to intelligently generate test code based on common vulnerability datasets. Detect if generative models create specific types of vulnerabilities without misbootstrapping as much as possible.
3) A statistical model for normalized efficiency evaluation was proposed.

II. Background

A. Large Language Models

Typically, Large Language Model (LLM) refers to a Transformer language model that contains billions or more of parameters, which are trained on a large amount of textual data, such as GPT-3 [1], PaLM [2], Galactica [3], and LLaMA [4]. LLM demonstrates strong understanding of natural language and the ability to solve complex tasks through text generation.

Thanks to Tian Xie for his great help in ChatGPT script generation

LLM exhibits some surprising emergent abilities, which may not have been observed in previous smaller PLMs. These abilities are crucial for language models to perform in complex tasks, making artificial intelligence algorithms unprecedentedly powerful and effective.

Nowadays, LLM is having a significant impact on the artificial intelligence community, and the emergence of ChatGPT and GPT-4 [5] has sparked a rethinking of the possibility of universal artificial intelligence (AGI). The rapid development of LLM is completely changing the research field of artificial intelligence. In the field of NLP, LLM can serve as a universal language task solver (to some extent), and the research paradigm has shifted towards the use of LLM. In the IR field, traditional search engines are challenged by new information search methods using AI chatbots (i.e. ChatGPT), and New Bing is the first to attempt to enhance search results based on LLM. In the field of CV, researchers have attempted to develop a visual language model similar to ChatGPT, which can better serve multimodal dialogues, and GPT-4 supports multimodal input by integrating visual information. This new technological wave may bring about a thriving real-world application ecosystem based on LLM.

B. Code Generation

In recent years, with the development of artificial intelligence, researchers have begun to explore a different path, which is to generate code based on pretrained large-scale language models. The basic idea of this method is to pretrain natural program code to obtain a large language model that can understand the code. Based on such a language model, natural language requirement descriptions or other prompts can be used to directly generate program code that meets the requirements through pretrained code models.

This method first pretrains the model through natural language text and one or more programming language codes to generate a pretrained language model. By fine-tuning the model on task specific data, the generation model for specific code generation tasks can be obtained. By generating a large number of code samples through this model, the correct code can be selected from the large number of samples through some post-processing program and used as the final generation result.

C. Code Security

Recently, the number of software vulnerabilities has rapidly increased, either publicly reported through CVE (Common Vulnerabilities and Exposures) or discovered internally in proprietary code. In particular, the popularity of open source libraries is not only due to incremental reasons, but also to the spread of influence. These vulnerabilities are mostly caused by insecure code and can be used to attack software systems, causing significant economic and social losses.

Vulnerability identification is a critical but challenging issue in the field of security. In addition to classic methods such as static analysis, dynamic analysis, and symbol execution, there has also been significant progress in applying machine learning as a supplementary method. In these early methods, machine learning algorithms used manually crafted features or patterns by human experts as input to detect vulnerabilities. However, the root cause of vulnerabilities varies depending on the vulnerability and the type of library, so it is impractical to describe all vulnerabilities in many libraries using handcrafted functionality. In order to improve the usability of existing methods and avoid the heavy workload of human experts in feature extraction. However, all of these works have significant limitations in learning comprehensive program semantics to describe highly diverse and complex vulnerabilities in real source code. Firstly, in terms of learning methods, they either view the source code as a flat sequence similar to natural language, or only represent it with partial information.

However, source code is actually more structured and logical than natural language, and has heterogeneity in representation, such as Abstract Grammar Tree (AST), data flow, control flow, etc. In addition, vulnerabilities can sometimes be subtle defects that require comprehensive investigation from multiple semantic dimensions. Therefore, the flaws in the design of previous works limited their potential to cover various vulnerabilities. Secondly, in terms of training data, some of the data in are labeled by static analyzers, which introduces a high percentage of false positives that are not true vulnerabilities. The other part is simple human code (even if there is ”good” or ”bad” in the code to distinguish between vulnerable and non vulnerable code), which far exceeds the complexity of real code.

III. Related Works

Existing studies on large language model (LLM) code generation predominantly center around mainstream languages. Pearce et al. (2022) and Majdinasab et al. (2023) explored Copilot’s performance, revealing a consistent but decreasing rate of insecure code suggestions [6] [7].Prenner et al. (2022) demonstrated Codex’s efficacy in fixing vulnerabilities, particularly in Python [10]. Asare et al. (2024) compared Copilot with human programmers, highlighting varying performance across different vulnerabilities [8].Xu et al. (2022) evaluated diverse LLMs, emphasizing the superiority of models trained on both natural language and code text [11]. Our work addresses a gap in research by focusing on both the security of LLM-generated code in the mainstream language C and that in the domain-specialized language solidity. We evaluate ChatGPT-3.5 using CWE and Confuzz solidity vulnerability datasets.

IV. Secode Framework

The effective operation of code generated by large models is crucial due to its widespread applications. However, accurately assessing the quality of code generated by large models remains an unresolved challenge. In response to this, we propose the Secode detection framework, designed to facilitate a precise evaluation of a certain language through the analysis of a substantial amount of generated code. The Secode framework encompasses three dimensions: correctness

evaluation, security assessment, and efficiency evaluation. The correctness assessment is responsible for verifying whether the generated code achieves its intended objectives. The security evaluation focuses on determining whether the generated code can run securely. Meanwhile, the efficiency evaluation aims to validate the operational overhead and performance of the code. In the subsequent sections, we provide a detailed exposition of each evaluation dimension, along with universally applicable implementation strategies.

A. Correctness Evaluation

Correctness evaluation in the context of large model-generated code comprises two integral components: compilation testing and functional testing. Compilation testing ensures the generated code can be successfully executed, while functional testing verifies if the code operates or interacts as intended by the user. In the specific implementation, the initial step involves generating a substantial volume of representative code tailored to a specific programming language. Subsequently, the generated code undergoes compilation testing using the language's respective compilation tools. For non-interactive code, direct execution is performed, and the results and intermediate outputs are examined for conformity to expectations. However, detecting correctness in interactive code poses additional challenges. To ensure the precision of testing, comprehensive simulations of all possible inputs are recommended. Here, we propose the adoption of FUZZ technology, which is applicable to various programming languages and has been previously implemented in multiple works, resulting in the development of diverse automated testing tools.The process diagram is shown in 1.

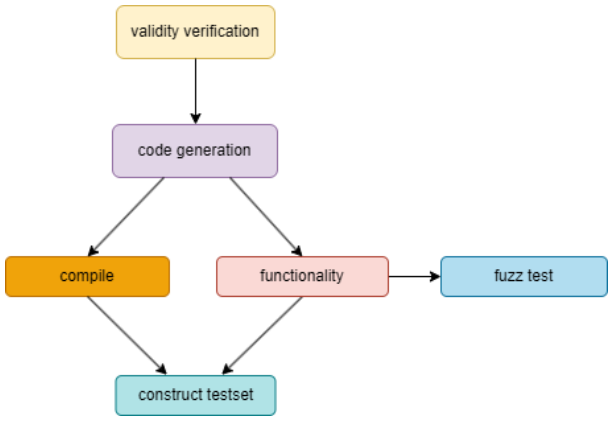

Fig. 1. Process description for correctness evaluation

B. Security Assessment

Security assessment stands as the most crucial and challenging aspect within the entire detection framework. Secode leverages vulnerability datasets of specific programming languages as inputs to reverse-generate code with original functionality using large models. The detection of vulnerability repair rates serves as an evaluation metric for the generative capacity of large models in the targeted programming language. The Security Assessment phase consists of four key steps: constructing the vulnerability dataset, extracting vulnerability features, reverse code generation, and code evaluation.

During the construction of the vulnerability dataset, we recommend labeling different vulnerability types for subsequent statistical evaluations. The extraction of vulnerability features involves describing critical information about vulnerabilities, with the specific implementation influenced by the chosen reverse code generation approach. Secode introduces two reverse code generation schemes: code regeneration and code completion.For code regeneration, the vulnerability feature extraction involves using the original vulnerable code as input to describe its functionality through a mechanism such as a large language model. For code completion, the vulnerability feature extraction involves removing the code containing vulnerabilities from the original code and adding completion prompts in the corresponding positions. Compared to code completion, code regeneration can better shield the impact of original vulnerability data on the reverse-generated code and is applicable to vulnerabilities triggered by global code structures. However, code completion better simulates real-world usage scenarios, thus it also possesses some evaluation value.

To ensure the validity of assessment conclusions, special attention is required during the code regeneration process to isolate original vulnerability information from the large model, ensuring that input data does not mislead the model. Notably, we emphasize that code-description and description-code transformations during code regeneration need to occur in different contextual environments.

Finally, a comprehensive evaluation of the large volume of code reverse-generated by the model is necessary. After a correctness evaluation of the obtained code, statistical assessments are performed through manual inspection or automated methods. Secode focuses on two metrics: vulnerability repair rates and the variety of generated vulnerabilities. The vulnerability repair rate represents the proportion of generated code without vulnerabilities to the total correct-generated code, providing an intuitive measure of the security of code generated by the large model. It can be calculated as follows.

$$Rate_{repair} = \frac{Fixedcode}{Correct - generatedcode} \tag{1}$$

The variety of generated vulnerabilities investigates whether the large model inadvertently introduces new vulnerabilities. Surprisingly, our research reveals that introducing new vulnerabilities is a prevalent phenomenon, making it a significant criterion for evaluating large model code generation. Upon completing these steps, the Secode assessment report is generated.The process diagram is shown in 2.

C. Efficiency Evaluation

The efficiency assessment includes two main dimensions: operational overhead and performance characteristics. Operational overhead refers to the extra computational resources needed for running the generated code. To assess operational

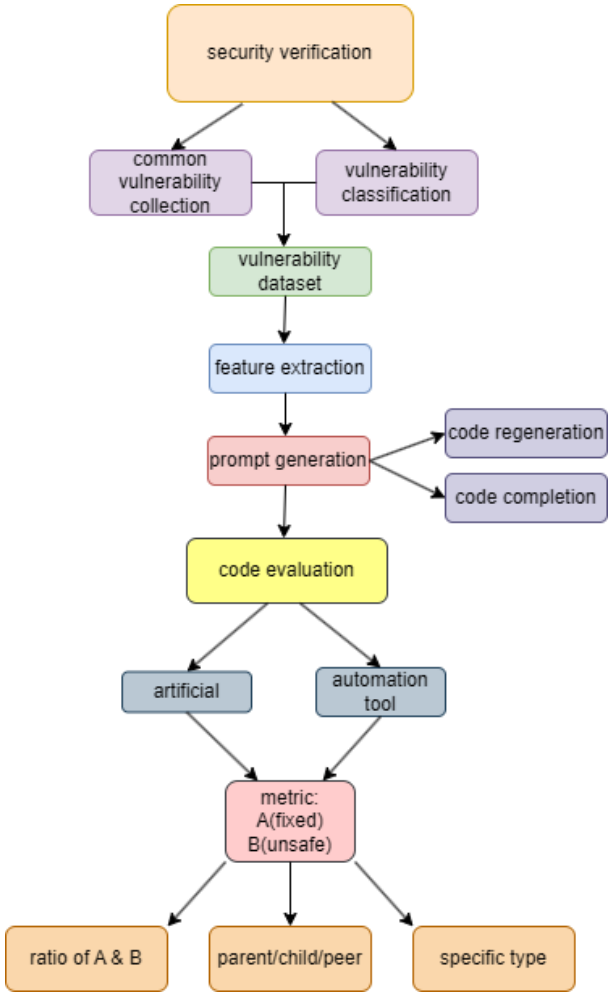

Fig. 2. Process description for security assessment

overhead, we need to select appropriate benchmark tasks for various programming languages and follow these steps:

- *Resource Utilization Measurement:* Measure the consumption of computational resources during the execution of the generated code. This includes monitoring CPU usage, memory utilization, and disk I/O.
- *Execution Time Analysis:* Analyze the time taken by the generated code to execute and compare it against baseline execution times for the task. This provides insights into the efficiency of the code in terms of execution speed.

Performance characteristics center on the responsiveness and scalability of the code produced. Various programming languages have distinct focuses, and Secode suggests considering the following metrics for testing:

- *Scalability Testing:* The ability of the generated code to scale with increasing workloads. This involves executing the code under varying input sizes and analyzing its performance.
- *Response Time Measurement:* Measure the response time of the generated code for different input scenarios, providing insights into its real-time responsiveness.
- *Concurrency Analysis:* Evaluate the code's concurrency capabilities by examining its performance under concurrent execution conditions. This involves assessing the efficiency of parallel processing and resource allocation.

V. Framework Empirical Test

In order to substantiate the universality and efficacy of the Secode detection framework, we conducted empirical experiments as follows. To assess the practical performance of Secode, we systematically selected representative programming languages from mainstream and domain-specific contexts for dedicated testing.

A. Evaluation for Mainstream Language Generation

The topic of code generation by large models for mainstream programming languages has long been acknowledged, with numerous studies focusing on evaluations for specific mainstream programming languages. In this research, we selected the highly representative C language as the test language to conduct a thorough assessment. Additionally, we utilized the widely adopted ChatGPT 3.5 model as the test model, recognizing its popularity and effectiveness in large-scale language generation tasks. We tested a wide range of vulnerability types in CWEs, and conducted several repetitive experiments to ensure data stability.

Describe the test plan in detail

B. Evaluation for Domain-specialized Language Generation

The quality assessment of code generation by large models for domain-specialized programming languages is often overlooked. However, with the advancement of AI technologies, the impact of large model code generation extends extensively to domain-specialized programming languages. Based on this, our research places significant emphasis on the evaluation of domain-specialized language. In our research, we choose Solidity as the testing language, which is motivated by the flourishing momentum of blockchain development and the widespread application of large model code generation in contract writing. Solidity's selection is also guided by its robust community and ecosystem, offering abundant tools and documentation support. This choice streamlines the quality assessment process within the Solidity framework.

Building upon the Confuzz dataset, we curated a novel dataset comprising 771 instances of common Solidity vulnerabilities. The vulnerabilities were categorized into 10 distinct classes and meticulously labeled. Subsequently, we conducted comparative tests using two code reverse-generation approaches: code regeneration and code completion. For code regeneration, we crafted two prompts designed to generate descriptions of vulnerability code and regenerate code based on those descriptions. During the experimental process, we strategically modified the prompts to avoid disclosing any information about the vulnerabilities during the code description generation. Additionally, we isolated the two processes to mitigate the potential misleading impact of original vulnerability information on code generation. Prompts are as below.

Listing 1. Code Description Prompt

```
# Prompt_for_Description
prompt_description = '''
Summarize the functionality of the given
    Solidity contract, providing details on
    all implemented functions along with their
     respective functionalities. Format the
    output as follows to ensure it can be
    effectively used to reconstruct the
    original code:

Primary functionality: ...
1. Function name, Functionality;
2. Function name, Functionality;
...

Note: Do not include any information regarding
     vulnerabilities in the output. Exclude
    any mention of vulnerabilities or comments
     related to security issues in the output.
'''
```

Listing 2. Code Regeneration Prompt

```
# Prompt_for_Regeneration
prompt_regen = '''
I want you to act as a blockchain developer.I
    will provide you with a description of the
     functionality of a Solidity code along
    with a summary of its code functions. The
    information format is as below:

Primary functionality(Contract Functionality
    Overview): ...
1. Function name, Functionality;
2. Function name, Functionality;

Please generate complete and executable
    Solidity code based on the provided
    information. Your response should be and
    only be pure codes, without any comments
    or annotations.
Your response should restrictly follow the
    format below:

pragma solidity ...
<code here (every function should be complete
    and exclude any annotation)>
Note: provide a full code without omitting any
     details.
Ensure that all aspects of the function,
    including input validation, logic, and
    return statements, are fully included in
    the generated code.
'''
```

Listing 3. Code Completion Prompt

```
# Prompt_for_Completion
prompt_comple_gen = '''
I want you to act as a blockchain developer. I
    will provide you with a piece of Solidity
     code that has a missing function which
    needs to be completed. The missing part is
     labeled as below: <!TODO:You need to
    complete this function>

Let's do it step by step :
1. Find the missing part which is labeled as
    <//TODO:You need to complete this function
    >
2. Complete the missing part based on the
    context and function name prompt.
3. Then output the completed function and only
     the missing part. Do not add any
    annotations, only the pure code is needed.

Let's do it!. Only return the full filled
    function including the function name.


'''
```

In the case of code completion, we initially removed specific details of vulnerable functions and any related vulnerability comments from the code. Placeholder identifiers ("¡// TODO: You need to complete this function¿") were inserted at the corresponding positions to prompt ChatGPT for code completion. To ensure output stability, we mandated that ChatGPT only returns the completed code snippets. Then we developed a program to seamlessly reintegrate the completed portions back into the deleted code.

During the code evaluation phase, we opted for an automated detection tool. Leveraging ConFuzz, which offers a fuzzing solution applicable to large-scale Solidity testing with 98.89% precision and 93.69% accuracy, we established it as our benchmark detection tool due to its remarkable accuracy. We conducted comparative tests and data analysis for both code generation approaches, which will be elaborated in detail in VI.

VI. Statistical Evaluation

A. Assessment of ChatGPT-C Generation

It is known that ChatGPT gave usable code in about 83% of the scenarios, safe code in about 40% of the scenarios, and safe code without vulnerabilities in about 48% of the scenarios with correct code. We analyze that ChatGPT does a excellent job of giving logically correct code suggestions that will pass compilation, but it is not satisfactory in terms of mainstream language code security, even though according to Sysnopsys' OSSRA report in 2022, 81%(2,049) of the codebases (human programmers) contain at least one vulnerability, while 49% of the codebases contain at least one high-risk vulnerability (Sysnopsys, 2022), we still expect ChatGPT to be more secure in code generation. It is worth noting that ChatGPt responds differently to various vulnerabilities, e.g., CWE-787 vulnerability is only about 17% fixed while CWE-79 vulnerability is 60% fixed, which should be taken into consideration when using ChatGPT as a code generation suggester.

B. Assessment of ChatGPT-Solidity Generation

For the solidity language, the overall performance of ChatGPT is not that good. The code suggestions given by it are vulnerable in about 60% of the cases. We find that ChatGPT's performance in the face of complex code environments(e.g. real-world code projects) is still quite flawed overall.

In Solidity Scenario One, we use an approach that gives prompts for ChatGPT to generate the corresponding code. We use a code security dataset from solidity. It contains 10 common vulnerability scenarios, each with a different number of test cases. The generated code was put through a code inspection tool. The average code coverage of the detection is about 90% and the average branch coverage is about 85%. We can believe that the detection basically contains code that may be vulnerable.

The detection results show that the code suggestion given by ChatGPT-3.5 replicated the original vulnerability in about 25% of the cases and fixed the original vulnerability in up to 75%, which indicates its good ability to fix known vulnerabilities. However, it is worth noting that it introduced other new vulnerabilities unrelated to the original vulnerability in about 33% of the cases, and the introduction of such new vulnerabilities greatly increased the vulnerability rate, thus reducing the credibility of ChatGPT's code security fixing efforts.

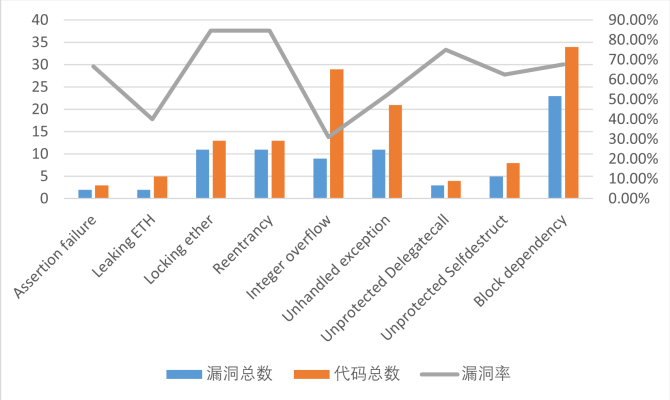

TABLE I
VULNERABILITY RATE

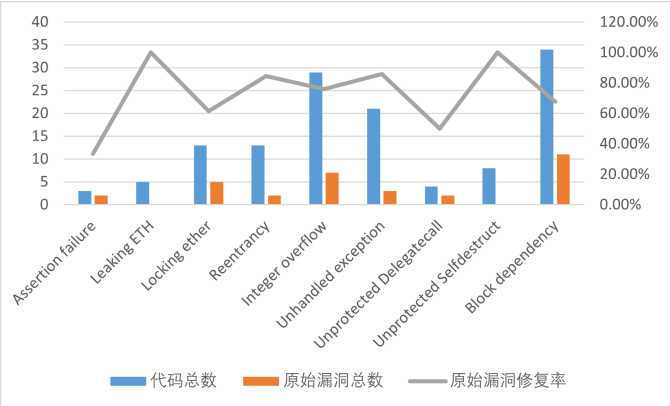

TABLE II
ORIGINAL VULNERABILITY FIX RATE

Meanwhile, we find that the code suggestions given by ChatGPT vary greatly for different vulnerability scenarios, with a fix rate of more than 50% for the vast majority of given vulnerabilities, and even 100% for the Unprotected Selfdestruct vulnerability. But for the Assertion failure vulnerability, the rate decreased to about 33%. It does a good job of fixing the code for a given vulnerability and also shows a preference for fixing different vulnerabilities.

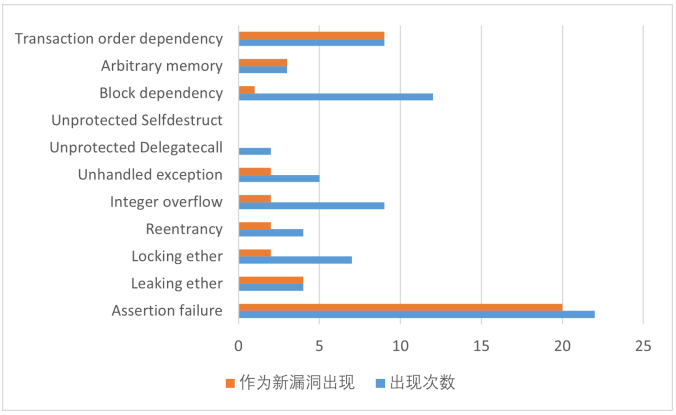

TABLE III
NEW VULNERABILITY

ChatGPT's code suggestions also show extreme differentiation in terms of examining the introduction of new vulnerabilities during the fixing process; some vulnerabilities are never introduced as new vulnerabilities, such as the Unprotected Delegatecall vulnerability and the Unprotected Selfdestruct vulnerability, some vulnerabilities are introduced as new vulnerabilities in almost any program, such as the Assertion failure vulnerability, and some vulnerabilities are not part of the test set but are introduced and detected by ChatGPT, such as the Arbitrary memory vulnerability. About 58% of the vulnerabilities were introduced as new vulnerabilities, meaning that the number of new vulnerabilities introduced in the code suggestions given by ChatGPT somewhat exceeded the number of original vulnerabilities it replicated.

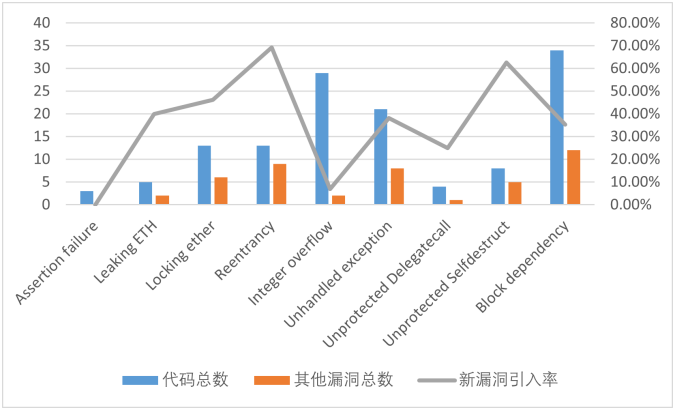

TABLE IV
NEW VULNERABILITY INTRODUCTION RATE

In Solidity Scenario Two, the approach we used was to give partial contexts for ChatGPT to patch the code. It turns out that there is a more significant difference compared to

Original Vulnerability	Gen Vul	CT Amount	Vul Rate	Fixed Rate	Intro Rate
Assertion failure	1	3	66.67%	33.33%	0.00%
Leaking ETH	6	5	40.00%	100.00%	40.00%
Locking ether	1,3	13	84.62%	61.54%	46.15%
Reentrancy	1,2,4,9,10	13	84.62%	84.62%	69.23%
Integer overflow	1,3,5	29	31.03%	75.86%	6.90%
Unhandled exception	2,4,6,10	21	52.38%	85.71%	38.10%
Unprotected Delegatecall	7,10	4	75.00%	50.00%	25.00%
Unprotected Selfdestruct	1,2,3,5	8	62.50%	100.00%	62.50%
Block dependency	1,2,4,5,9	34	67.65%	67.65%	35.29%
Avg Vul Rate		59.23%	**Avg Code Coverage**		89.96%
Avg Origin Fixed Rate		75.38%	**Avg Intro Rate**		34.62%
1 Assertion failure 2 Leaking ETH 3 Locking ether 4 Reentrancy 5 Integer overflow 6 Unhandled exception 7 Unprotected Delegatecall 8 Unprotected Selfdestruct 9 Block dependency 10 Transaction order dependency					

TABLE V
SOLIDITY HINT GENERATION METHOD

Original Vulnerability	Gen Vul	CT Amount	Vul Rate	Fixed Rate	Intro Rate
Assertion failure	0	3	00.00%	100.00%	0.00%
Leaking ETH	1,2,5,6	5	80.00%	80.00%	60.00%
Locking ether	1,3,5,11	12	91.67%	50.00%	41.67%
Reentrancy	1,2,5,9,10	12	91.67%	100.00%	100.00%
Integer overflow	1,3,5,8	21	85.71%	42.86%	28.57%
Unhandled exception	1,2,3,4,6,9,10	16	68.75%	62.50%	68.75%
Unprotected Delegatecall	1,7	5	100.00%	00.00%	40.00%
Unprotected Selfdestruct	1,2,4,6,8,9,11	10	90.00%	60.00%	100.00%
Block dependency	1,2,3,4,5,9,10	34	91.18%	58.82%	58.82%
Avg Vul Rate		84.75%	**Avg Code Coverage**		75.32%
Avg Origin Fixed Rate		59.32%	**Avg Intro Rate**		58.47%
0 No Vulnerability 1 Assertion failure 2 Leaking ETH 3 Locking ether 4 Reentrancy 5 Integer overflow 6 Unhandled exception 7 Unprotected Delegatecall 8 Unprotected Selfdestruct 9 Block dependency 10 Transaction order dependency 11 Arbitrary memory					

TABLE VI
SOLIDITY COMPLEMENT GENERATION METHOD

the first set of data, where the average vulnerability rate rises to about 85%, while the average original vulnerability fixing rate drops to about 60%, and at the same time unrelated new vulnerabilities are introduced in about 60% of the cases. The insecurity of the same dataset rises almost significantly in this application scenario, and we hypothesize that this usage scenario is not a good one for ChatGPT-3.5, since its generated code is somewhat influenced by the original test code given. We conclude that the first scenario, i.e., giving only the prompt about requirements, is a better representation of the security of the code suggestions given by LLM, which can somewhat shield the generated code from the influence of the original vulnerability data. However, the second scenario also has practical applications in reality, for which the safety of LLM-given code should be carefully evaluated, and a better way to use it is to use it only in the first scenario.

We then came to some conclusions that ChatGPT can do a better job in code security when only code requirements are given rather than a specific code context. ChatGPT performs well in handling the fixing of known vulnerabilities, although it shows a preference for different vulnerabilities, and it is able to handle the given vulnerability in most of the cases. However, its ability to fix vulnerabilities in real-life complex environments deserves to be questioned and further evaluated, as there is a not-so-small chance that it will introduce one or several unrelated new vulnerabilities while fixing the original vulnerability, which may be caused by its lack of a holistic view of a code project.

VII. CONCLUSION

REFERENCES

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. *Language Models are Few-Shot Learners.* **https://doi.org/10.48550/arXiv.2005.14165**

[2] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, et al. *Palm: Scaling language modeling with pathways.* **https://doi.org/10.48550/arXiv.2204.02311**

[3] R. Taylor, M. Kardas, G. Cucurull, T. Scialom, A. Hartshorn, E. Saravia, A. Poulton, V. Kerkez, and R. Stojnic, *Galactica: A large language model for science.* **https://doi.org/10.48550/arXiv.2211.09085**

[4] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. Lachaux, T. Lacroix, B. Roziere, N. Goyal, E. Ham-bro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. *Llama: Open and efficient foundation language models.* **https://doi.org/10.48550/arXiv.2302.13971**

[5] OpenAI *Gpt-4 technical report.* OpenAI, 2023.

[6] Pearce, H., B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri 2022, May.*Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions.*In 2022 *IEEE Symposium on Security and Privacy* (SP), pp.754–768. ISSN: 2375-1207.*IEEE Symposium on Security and Privacy* **https://doi.org/10.48550/arXiv.2108.09293**

[7] Majdinasab, V., M. J. Bishop, S. Rasheed, A. Moradidakhel, A. Tahir, and F. Khomh, 2023, Nov.*Assessing the Security of GitHub Copilot Generated Code – A Targeted Replication Study.*
https://doi.org/10.48550/arXiv.2311.11177

[8] Asare, O., M. Nagappan, and N. Asokan, 2024, Jan. *Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code?* in *Empirical Software Engineering.*
https://doi.org/10.48550/arXiv.2204.04741

[9] Pearce, H., B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, 2022, Aug.*Examining Zero-Shot Vulnerability Repair with Large Language Models* in 2023 *IEEE Symposium on Security and Privacy (SP).*
https://doi.org/10.48550/arXiv.2112.02125

[10] Prenner, J. A., H. Babii, and R. Robbes, 2022, Oct.*Can OpenAI's codex fix bugs?: an evaluation on QuixBugs* in *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*
https://doi.org/10.1145/3524459.3527351

[11] Xu, F. F., U. Alon, G. Neubig, and V.J. Hellendoorn 2022, June. *A systematicevaluation of large language models of code.* In Proceedings of *the 6th ACMSIGPLAN International Symposium on Machine Programming*, San Diego CA USA, pp. 1–10. ACM.
https://doi.org/10.48550/arXiv.2202.13169

[12] Pearce, H., B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, 2022, Jun.*An Empirical Evaluation of GitHub Copilot's Code Suggestions* in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*
https://doi.org/10.1145/3524842.3528470