International Journal of Enhanced Research in Science, Technology & Engineering
ISSN: 2319-7463, Vol. 11 Issue 7, July-2022, Impact Factor: 7.957

Utilizing Large Language Models to Enhance Detection and Management of Software Vulnerabilities and Cybersecurity Threats

Geetha Priya D T[1], Dr. Rajesh Koolwal[2]

[1]Research Scholar, Department of Computer Science, JS University, Shikohabad, Uttar Pradesh
[2]Assistant Professor & Supervisor, Department of Computer Science, JS University, Shikohabad, Uttar Pradesh

ABSTRACT

With its potential for identifying and mitigating security risks, Large Language Models (LLMs) have become formidable instruments in the field of software vulnerability and cybersecurity jobs. Vulnerability identification, threat prediction, and automatic code repair are just a few of the cybersecurity applications of LLMs that are discussed in this article. We define LLMs, discuss their many uses, and then use a literature study to assess their merits and shortcomings. We demonstrate the efficacy of several LLMs in various cybersecurity fields by investigating their performance on tasks such as code summarization and virus identification. Our research shows that LLMs outperform conventional methods when it comes to finding security flaws and suggesting solutions. Our focus is on the workflow of LLM models and how they may be integrated into incident response systems and frameworks for detecting cyber threats. We also go over some supplementary techniques and resources, such static and dynamic code analyzers, that can improve LLMs' performance. Furthermore, we compile results from prior studies to show how LLMs have greatly improved efficiency in uncovering and fixing software vulnerabilities and cybersecurity risks. Last but not least, drawing on previous research, the report provides recommendations for improving LLM deployment.

INTRODUCTION

Software programs can exhibit implementation faults in the form of program crashes, corrupted data, decreased performance, and exploitable security vulnerabilities. This highlights the importance of finding and fixing bugs as soon as possible. Both traditional methods, like static analysis, and more modern ones, such deep learning-based approaches, have their limits when it comes to effectively detecting software flaws. In the early phases of bug discovery, static analysis and DL approaches provide clear benefits; however, they have a number of limitations, such as a high rate of false positives and difficulties with interpretability and feature selection. The efficiency of developers can be severely hindered by these false positives.

A lot of people in the software engineering field are interested in the new deep learning (DL) developments and how they could fix old problems. To fix bugs in programs, several have suggested using Deep Learning (DL) methods. Notable examples that employ DL to directly fix these issues include DeepRepair [49] and DLFix [48]. CURE [50] uses a deep learning model that has been trained on a corpus of source code to understand and recreate the structure of code. DEAR [51] employs a mixed strategy, integrating deep learning with spectrum-based fault localization, to improve code context awareness. To find the best ways for DL-based program defect reduction, additional study is needed to evaluate the pros and cons of each of these methods.

However, Natural Language Processing (NLP) has been rocked by the advent of Large Language Models (LLMs) driven by the groundbreaking Trans-formers architecture. This has brought about a new age of capabilities and applications, leading to remarkable improvements. A number of recent research have shown that LLMs are capable of discovering security vulnerabilities more effectively than both traditional approaches and deep learning (DL). [9]

Problem Statement (challenges)
Organizations' and individuals' ability to defend themselves is called into question as cyber threats become more complicated and severe at an alarming rate. On average, it takes 197 days for firms to notice a security breach and 69 days to contain it,

according to the 2023 Verizon Cost of Data Breach Report. Longer incident reaction times put businesses at risk of operational and financial losses due to things like unanticipated outages and lower productivity. Computers are now absolutely necessary for a wide variety of linguistic tasks, including cybersecurity (such as finding bugs in software code) and natural language communications.

The necessity for automated techniques to identify software defects is growing due to the inefficiency of traditional vulnerability detection approaches that rely on human specialists identifying weaknesses. In many domains, machine learning approaches clearly outperform traditional methods, with better detection rates and more practical solution suggestions. False positive or negative results could result from factual hallucinations or solution misinterpretations, which even LLMs are not immune to. Consequently, a strong and all-encompassing system for finding and resolving software vulnerabilities and security threats still necessitates substantial study into the development of a unified approach that incorporates LLMs with other techniques.

Research goals
This research primarily aims to investigate the possible uses of LLMs in the field of cybersecurity threat identification and software vulnerability analysis. Providing researchers with basic information on LLMS and its practical uses is our secondary goal. After reviewing the literature, we will offer a detailed set of guidelines and recommendations to help researchers use LLMs to find and fix cybersecurity and software vulnerabilities. We will also discuss the difficulties of using LLMs for software vulnerability and cybersecurity threat detection and provide solutions to these problems. Our goal in taking on these problems is to make software development safer and more effective by releasing LLMs to their full potential.

Overview
The next parts detail our methodology, which includes how we gathered and analyzed our sources. Next, we have the introduction, which sets the stage for the rest of the paper by outlining the history, classification, and uses of LLMs in software engineering and cyber security.

Before we get into our research aims and objectives, we lay out the problems with using LLMs for software vulnerability management and cybersecurity. The next step is to formulate the research questions. Following a brief overview of the literature, we dive into the meat of our research and show how Large Language Models (LLMs) are used to manage cybersecurity risks and software vulnerabilities. After then, the research questions are addressed and answered in a systematic manner.

METHODOLOGY

A systematic literature evaluation was carried out to thoroughly investigate the state of the art regarding the use of Large Language Models (LLMs) for the purpose of detecting and managing software vulnerabilities and cybersecurity risks. The methodology that was used is outlined in the following steps:

Finding Relevant Terms:
With the initial set of keywords being "LLMs" and "cyber security," we began our investigation. The following keywords were discovered after reviewing the connected articles; they were utilized to locate further articles and direct the search:

Code Languages, Pre-trained Transformers, Cyber Threat Detection, Software Fix, Security Testing, Penetration Testing, Software Vulnerability detection, ChatGPT, GPT-4, Bard, Code Generation, Vulnerability Discovery, transformer, overfitting, prompt, and code analysis are all part of the larger language models.

Google Scholar search:
A variety of articles pertaining to LLMs, software vulnerabilities, and cybersecurity concerns were retrieved from Google Scholar searches that utilized the specified keywords.
Examination of the Found Results:

Articles that fulfilled the research goals were located by carefully examining the search results, paying special attention to the titles, abstracts, and keywords.

Method of Snowballing:
We used the snowball method by looking for more sources that could be relevant to the study in the references of the articles we chose to read.

Continuous Improvement:
To make sure the literature review was comprehensive and up-to-date, the search keywords were adjusted iteratively.

Criteria for Inclusion and Exclusion:
Articles were included or excluded based on the following criteria:

Sources that do not help answer the study's questions or accomplish its goals are considered irrelevant. Excluding sources that are not published in English, unless there is a strong focus on language diversity, are non-English publications. Unfinished Information: Removing sources from consideration if they do not provide complete or clear information pertinent to the research.

Blog entries and promotional materials are examples of non-professional or non-academic sources; yet, they may contain unique insights.

Analyzing critically:
We conducted a thorough evaluation of the chosen articles, aiming to glean important insights, methods, and results. To make sure strong and applicable contributions were included, the research was evaluated for quality.
Combination and Cohesion:

The following portions of the research report were built around the information synthesised from the chosen publications, which sought to find common themes, trends, and gaps in the current literature.

Applying AI Support
If any questions remained after going over all the articles, we used Microsoft Copilot to pose them. Afterwards, we consulted articles on arXiv.org to discover the answers to the following questions:

If you know of any articles on arXiv.org that can shed light on this subject, please provide them. So, what is the deal?

Using this strategy, we were able to supplement our research with the help of artificial intelligence and the vast resources available on arXiv.org. Referencing and Citation: The chosen articles were referenced in the study using the ACM citation style, which guarantees correct attribution of the sources.

BACKGROUND

Language Models for Large Scale Data (LLMs)
An LLM, what is it? One kind of language model that is trained using a significant amount of text data is the significant Language Model, which is also known as LLMs. A few examples of LLMs are RoBERTa, GPT-4, and BERT [23]. They represent the cutting edge of artificial intelligence models built on neural network architectures, especially deep learning, and have proven themselves capable across a range of Natural Language Processing (NLP) tasks. (23, 55). The "LLM" stands for large-sized PLMs and is used to differentiate language models according to their size. Since the capacity of the model is dependent on the overall amount of computational resources and the size of the data, there is no formal agreement on the minimum scale of parameters for LLMs.

Lauded for their sophisticated linguistic talents, large training datasets, and neural network architectures, LLMs have outstanding skill in producing and refining natural language. Various domains, linguistic styles, and subject areas are covered in the extensive corpus of texts used to train LLMs by developers. During training, LLMs develop a statistical understanding of language, which allows them to produce context-aware, cohesive text and perform exceptionally well on a variety of natural language processing tasks. Long short-term memories (LLMs) use massive neural networks with tens or hundreds of billions of parameters to generalize well across different language tasks and pick up on subtle semantic linkages. The size and design of LLMs enhance their ability to incorporate extensive language knowledge, demonstrating an artificial inventiveness that mirrors their training data [41].

Furthermore, LLMs are discovering uses in a wide range of practical domains, including content generation, chatbots, machine translation, and factual subject summarization. Even though these models have amazing capabilities, developers must be mindful of the possibility of biases in their training data and act responsibly while using them. There are continuous efforts to increase their capabilities, remove restrictions, and explore new applications in the field of LLM research, which is causing them to evolve rapidly.

Classification of LLMs

In order to give a general outline of LLMs and how they are classified, we rely on the extensive work done by [1, 3, 6, 12]. By classifying LLMs and illuminating their varied uses and traits, the authors have accomplished an admiral feat. Since its introduction and publication in 2017, the Transformer design has been utilized by the most powerful Large Language Models (LLMs) at present.[43] Within encoder-only, decoder-only, or encoder-decoder networks, transformer topologies can be utilized according to the application. The aforementioned study classifies the most popular LLMs as either encoder-only, encoder-decoder, decoder-only, or Sparse Model architecture [1–3].

When given a sequence of inputs, encoder-only models may grasp and summarize the data quite well. Instead of creating new sequences, they make representations of a predetermined size, which is ideal for tasks like categorization. Their capacity to produce results based on the acquired context restricts their use in situations needing further generation, but, they are effective in tasks requiring thorough comprehension, such as SoC security verification.

Specifically, LLMs that are encoder-only, such as BERT (Bidirectional Encoder Representations from Transformers) and its variants, aim to encode input sentences in order to capture contextual information and word relationships. BERT's bidirectional attention technique takes into account both the left and right context during training; it is built on transformer's encoder architecture. A number of specialized models have been developed with software engineering (SE) tasks in mind. These models include CodeBERT, GraphCodeBERT, RoBERTa, ALBERT, BERTOverflow, and CodeRetriever. These models incorporate program structures, new pre-training tasks, or utilize different modalities to enhance their application to code-related tasks. To help with things like code completion and bug discovery, CodeBERT enhances code comprehension by forecasting future tokens. GraphCodeBERT improves code summarization and program analysis by seeing the relationships between code elements as graphs [1]. Code review, analyzing bug reports, and named entity recognition within code entities are tasks that these models excel at.

LLMs with encoder and decoder

Reversible encoder-demodulator A language model that uses an encoder and a decoder module is called a Large Language Model (LLM). In order to produce the desired output text, the encoder module must first encode the input sentence into a hidden-space. More flexible training tactics are made possible by this framework. The transformer, first presented in, is a famous example of an encoder-decoder system.

Models that demonstrate flexibility in tasks like summarization, translation, and question-answering include BART, PLBART, T5, CodeT5, AlphaCode, and CoTexT. Thorough and comprehensive research conducted in, where a comprehensive survey of Large Language Models was carried out.

SOFTWARE VULNERABILITIES DETECTION, A SUBCATEGORY OF SOFTWARE DEVELOPMENT LIFE CYCLE

As part of the Software Development Life Cycle (SDLC), Large Language Models (LLMs) are used to identify and manage software vulnerabilities and security concerns. The six distinct phases that make up software development life cycle (SDLC) will help shed light on the function of LLMs within this framework.

Requirements engineering
Software design
Software development
Software quality assurance (include software vulnerabilities)
Software management
Software maintenance (include cyber security threats)

Stages 4 (Software Quality Assurance) and 6 (Software Maintenance) of the Software Development Life Cycle (SDLC) are where Large Language Models (LLMs) mostly help in finding and managing software vulnerabilities and cybersecurity risks. The first three phases of a software development life cyclerequirements engineering, software design, and development also make use of LLMs.

We will thoroughly examine the role of LLMs in phases 4 and 6, which will constitute the major emphasis of this study. To provide a comprehensive understanding of the LLM's position in SDLC, stages 1, 2, 3, and 5 will each include a brief introduction and examples. Stages 4 and 6 will be thoroughly examined in the parts that follow.

Utilization of LLMs in Requirements Engineering

For many purposes pertaining to software requirement planning and definition, Large Language Models (LLMs) are invaluable.

Clearing Confusion

By deciphering the true meaning of vague software requirements, ChatGPT and ELECTRA make them much easier to comprehend.

Sorting Requirements

When it comes to breaking down requirements into functional and non-functional categories, BERT-based models excel. Gaining insight into the project at its first phases is facilitated by this.

Identifying Terms

Combining BERT-based approaches with K-means clustering, a grouping method, allows for the successful discovery of phrases with multiple meanings across diverse domains. Because it trains using permutations, XLNet has also demonstrated some success in this domain.

Finding Connections

Innovative applications of the BERT and T5 models in Requirement Engineering have yielded encouraging results in the discovery of relationships between entities.

Automating Traceability

T-BERT is a powerful tool for translating between programming language artifacts and natural language artifacts in order to facilitate traceability, which is defined as the capacity to track an item's usage, location, or history. Reliability and practicality in monitoring software and system changes are demonstrated by this. [1].

Utilization of LLMs in Software Design

The area of GUI retrieval makes use of learning-to-rank (LTR) models that are based on BERT. Natural language (NL)-based GUI ranking is used to assess the effectiveness of these models, which are challenged with ranking GUI documents based on text. Rapid prototyping is not complete without Large Language Models (LLMs), which contribute to the process by providing faster design methods [28]. With this, you have a methodical approach to solving problems with LLM for Software Engineering (LLM4SE). Furthermore, LLMs are utilized by the SpecSyn Framework to automate the synthesis of software specifications. According to the F1 score, which is a measure of test accuracy, this method has proven to be more effective than prior methods in both single and multiple sentence analysis.

Utilization of LLMs in Software Development

Codex, InCoder, CodeGeeX, CodeGPT, BERT, and other Large Language Models (LLMs) have become indispensable in software development for improving efficiency in a wide range of code-related activities, such as program analysis, interpretation, and generation. Launched as CodeX—the foundation for GitHub Copilot—the era of LLMs in code generation officially began. The efficacy of code creation has been further enhanced by subsequent models such as InCoder, Google Alphacode, and Amazon CodeWhisperer. An AI-powered chatbot known for its remarkable language recognition and answers that resemble human speech was recently released by OpenAI. The bot is called ChatGPT. Even though ChatGPT has been successful at addressing programming challenges and generating code accurately, top software businesses still have worries about the quality of the code it generates.

Code completion, automatic generation, and annotation-to-code conversion are all made easier with the use of LLMs, which can convert natural language descriptions into code. These models provide recommendations based on your specific needs, as well as code suggestions and semantic comprehension. Code readability, documentation, and comprehension are all enhanced by their capacity to generate human-readable descriptions from source code. In addition, they enhance code maintenance and integration efficiency by interpreting code comments, semantics, and dependencies.

The recommendations, synthesis, and documentation of APIs are likewise heavily reliant on LLMs. Models that can adapt to changes in documentation in real-time and create API calls appropriately include LLaMA-based architectures and GPT-4. Automated monitoring and enhancement of API documentation is made possible with their assistance in detecting and warning about poor API documentation quality. On top of that, LLMs can greatly improve program execution speed by suggesting changes to the code that boost performance. In addition to enhancing developer productivity, they provide code samples by utilizing open-source projects. Finally, they help with identifier normalization, which means that identifier vocabulary is aligned with real language and code comprehension is enhanced.

Utilization of LLMs in Software Management

Few academic articles have yet to detail how LLMs are being used in software management. When it comes to software maintenance work estimation, Alhamed et al. [52] test BERT's usefulness. While their research shows that BERT could help with decision-making and provide useful insights, it also shows that there are some problems with it and that more research is needed.

In software quality assurance, Large Language Models (LLMs) are leveraged across various tasks that are mostly related to software vulnerabilities detection. Such as:

Test Generation: Automated test case creation: LLMs aid in generating diverse test cases, improving coverage, and identifying potential defects [1].

Natural language-based test case generation

Collaboration between developers and testers is fostered by generating test cases from natural language descriptions [1].

Vulnerability Detection

LLMs by leveraging their understanding of code structures and semantics, contribute to detecting vulnerabilities, offering

Function-level vulnerability detection

improved accuracy compared to traditional methods

Fine-tuning models like BERT for vulnerability detection has proven effective, as seen in studies that combine sequence and graph embedding for function- level vulnerability detection [1].

Test Automation: LLMs enhance automated testing techniques like mutation testing and fuzzing. They introduce faults in the codebase to evaluate the effectiveness of test suites and generate diverse input programs to uncover vulnerabilities and bugs. Incorporating LLMs into testing techniques improves test coverage, detects bugs efficiently, and aids in building more robust software systems [1, 9]

Formal verification and repair

LLMs combined with formal methods assist in automatically repairing software based on formal verification.

Formal verification methods are bolstered by LLMs, particularly when combined with bounded model checking. These models can automatically repair software based on formal methods, showcasing their ability to understand intricate software structures and generate accurate repairs, ensuring stable and secure performance [1].

Bug Localization

Augmenting bug reports with token and paragraph-level operations and training BERT-based models with augmented data significantly enhances bug localization. These techniques expand the training data and improve models' precision in identifying the specific source code segments responsible for reported bugs or defects.

Failure-inducing Test Identification: LLMs can assist in identifying fault-inducing test cases by leveraging their understanding of expected behavior from erroneous programs. Combining LLMs with difference testing techniques helps pinpoint subtle code differences, leading to more accurate identification of fault-inducing test cases [1].

Flaky Test Prediction

Environments where test cases exhibit non-deterministic behavior, LLMs like Code BERT assist in predicting flaky tests. Their predictions aid developers in focusing debugging efforts on potentially problematic test cases, reducing human effort and execution time spent on debugging [1].

In the field of software maintenance, Large Language Models (LLMs) are extensively utilized for a variety of tasks. These tasks include, but are not limited to, the detection and mitigation of cybersecurity threats and software vulnerabilities.

Program Repair: LLMs like BERT, CodeBERT, and ChatGPT have shown effectiveness in generating accurate patches for bugs and defects. ChatGPT's interactive design enables continuous feedback loops, enhancing accuracy in program repair [1].

Code Review
LLMs aid reviewers in understanding code intent, detecting errors, and suggesting improvements, thereby enhancing code quality

Debugging
LLMs simulate scientific debugging processes, generate hypotheses about code problems, and debug their own generated code [1].

Bug Report Analysis
LLMs analyze bug reports, provide repair suggestions, and aid in better understanding the underlying causes, expediting error-fixing processes.

Code Clone Detection: BERT's application in code clone detection, particularly in identifiers, enhances clone detection across all layers [1]. Logging: Models like T5 assist in automatically generating and summarizing logs, aiding in understanding software behavior and identifying issues [1].

Bug Prediction and Triage
LLMs like BERT effectively predict long-lived bugs and assist in triaging bugs, improving error detection and resolution [1].

Bug Report Replay and Duplicate Detection
LLMs aid in automated error replay based on natural language understanding and are used for detecting duplicate bug reports, reducing redundancy in the software development process [1].

Decompilation and merge Conflicts Repair
LLMs like ChatGPT aid in recovering symbolic names during decompilation and partially automate program merge conflicts repair [1].

Sentiment Analysis and Tag Recommendation
Transformer models outperform existing tools in sentiment analysis related to software products and help recommend tags for software Q&A sites [1].

Vulnerability Repair and Traceability Recovery
Challenges persist in LLMs generating functionally correct code for zero-point vulnerability remediation. Additionally, LLMs enhance traceability link predictions, refining traceability recovery [1].

CONCLUSION

This article explored the potential of LLMs, examining their various applications across the Software Development Life Cycle (SDLC), particularly in the stages of software quality assurance and maintenance. We reviewed successful applications of LLMs in tasks like test generation, bug localization, code repair, and vulnerability detection. Our analysis revealed that LLMs, when properly trained and integrated with existing tools, can significantly enhance software security. LLMs can outperform traditional static code analyzers in vulnerability detection and propose potential fixes, reducing the burden on developers. Furthermore, their ability to process and understand natural language empowers them to analyze textual information from logs and reports, aiding in threat identification and incident response. However, it is crucial to acknowledge the limitations of LLMs.

As LLM technology continues to evolve, so too will its applications in software security. By addressing current limitations and implementing best practices for training and integration, LLMs have the potential to revolutionize how we approach software security, enabling the development of more secure and robust systems.

Future research directions include exploring methods to improve the explainability and transparency of LLM decision-making processes. Additionally, continuous learning techniques can be incorporated to ensure LLMs stay up-to-date with the ever-changing threat landscape. By bridging the gap between cutting-edge research and practical implementation, LLMs can become a cornerstone of a comprehensive software security strategy. We hope our work guides and inspires more research, encourages new ideas, and helps collaborations in this growing field.

REFERENCES

[1] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J.: Haoyu Wang. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv:2308.10620v4 2024

[2] Charalambous, Y., Tihanyi, N., Jain, R., Sun, Y., Ferrag, M., Amine, Cordeiro, L.: A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification. (2023). 10.48550/arXiv.2305.14752

[3] Dipayan Saha, S., Tarek, K., Yahyaei, S.K., Saha, J., Zhou: Mark Tehranipoor, Farimah Farahmandi. LLM for SoC Security A Paradigm Shift. arXiv:2310.06046v1 2023

[4] Ferrag, M.A., Ndhlovu, M., Tihanyi, N., Cordeiro, L.C.: Merouane Debbah, Thierry Lestable, Narinderjit Singh Thandi. Revolutionizing Cyber Threat Detection with Large Language Models. arXiv:2306.14263v2 2024.

[5] Andreas Happe and Jürgen Cito: Getting pwn'd by AI: Penetration Testing with Large Language Models. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023). Association for Computing Machinery, New York, NY, USA, 2082–2086. (2023). https://doi.org/10.1145/3611643.3613083

[6] Sakaoglu, S.: 'KARTAL: Web Application Vulnerability Hunting Using Large Language Models: Novel method for detecting logical vulnerabilities in web applications with finetuned Large Language Models', Dissertation, (2023)

[7] Ferrag, M.A., Battah, A., Tihanyi, N., Debbah, M., Lestable, T.: Lucas C. Cordeiro. SecureFalcon The Next Cyber Reasoning System for Cyber Security. arXiv:2307.06616v1 2023

[8] Weng, G., Andrzejak, A.: Automatic Bug Fixing via Deliberate Problem Solving with Large Language Models, in 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), Florence, Italy, 34–36. (2023). pp 10.1109/ISSREW60843.2023.00040

[9] David Noever: Can Large Language Models Find and Fix Vulnerable Software. arXiv:2308.10345v1 2023

[10] Hammond Pearce, B., Tan, B., Ahmad, R., Karri: Brendan Dolan-Gavitt. Examining Zero-Shot Vulnerability Repair with Large Language Models. arXiv:2112.02125v3 2022.

[11] Jingxuan He and Martin Vechev: Large Language Models for Code: Security Hardening and Adversarial Testing. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23). Association for Computing Machinery, New York, NY, USA, 1865–1879. (2023).

[12] https://doi.org/10.1145/3576915.3623175

[13] Xia, C.S., Wei, Y.: Lingming Zhang. Practical Program Repair in the Era of Large Pre-trained Language Models. arXiv:2210.14179v1 2022

[14] Purba, M., Ghosh, A., Radford, B., Chu, B.: Software Vulnerability Detection using Large Language Models, in 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), Florence, Italy, 112–119. (2023). pp 10.1109/ISSREW60843.2023.00058

[15] Yuqiang, S., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., Yang Liu:. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. arXiv:2308.03314v2 2023.

[16] Huang, K., et al.: An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair, 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), Luxembourg, Luxembourg, pp. 1162–1174, (2023). 10.1109/ASE56229.2023.00181

[17] Katsadouros, E., Patrikakis, C.Z., Hurlburt, G.: Can Large Language Models Better Predict Software Vulnerability? in IT Professional. May-June. **25**(3), 4–8 (2023). 10.1109/MITP.2023.3284628

[18] Marwan Omar: Detecting software vulnerabilities using Language Models. arXiv:2302.11773v1 2023

[19] Mamede, C., Pinconschi, E., Abreu, R., Campos, J.: Exploring Transformers for Multi-Label Classification of Java Vulnerabilities, 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), Guangzhou, China, 2022, pp. 43–52, 10.1109/QRS57517.2022.00015 enchmarks. arXiv:2312.12575v2 2024