Vulnerability Mapping and Mitigation Through AI Code Analysis and Testing

Tauheed Waheed[a], Eda Marchetti[b] and Antonello Calabrò[c]
CNR-ISTI Pisa, Italy

Keywords: Vulnerability Mapping, Cybersecurity, AI-Driven, Testing.

Abstract: The research addresses the significant and complex challenge of vulnerability mapping and repairing code vulnerabilities, which is critical for enhancing cybersecurity in our increasingly technology-driven society. This paper aims to present an in-depth methodology and framework for effectively mapping software vulnerabilities through AI-driven code analysis and testing techniques. The proposed method and framework provide an automated environment that facilitates identifying and mitigating security vulnerabilities. This innovative framework benefits prosumers and developers, empowering them to confidently produce secure code, even with inadequate cybersecurity knowledge or extensive testing experience. By leveraging AI, the methodology streamlines the process of vulnerability detection and enhances overall software security.

1 INTRODUCTION

The vulnerability analysis is an essential and standard procedure for identifying cybersecurity threats, risks, and deficiencies within a computing infrastructure (Fatima et al., 2023). This process aids organizations in proactively addressing security vulnerabilities, preventing cyber-attackers from exploiting them for financial gain or other destructive goals. However, over the years, particularly in the last two decades, the spectrum of security-related challenges has expanded significantly as internet and mobile computing technologies evolve. Therefore, it has become a vital task to develop comprehensive mitigation strategies and techniques that safeguard essential business operations (Waheed and Marchetti, 2023).

Over the years, researchers have proposed various vulnerability assessment methods, broadly categorized into manual, assistive, and fully automated approaches (Fatima et al., 2023) (Janovsky et al., 2024). Moreover, manual vulnerability assessments rely on experts (humans) who follow explicit instructions to identify security vulnerabilities. The method has various drawbacks; it is time-consuming, it is resource-intensive and heavily dependent on specialized knowledge, which is often in short supply.

[a] https://orcid.org/0009-0006-0489-7697
[b] https://orcid.org/0000-0003-4223-8036
[c] https://orcid.org/0000-0001-5502-303X

Assistive vulnerability assessments utilize scanning tools or frameworks typically updated to detect relevant security weaknesses (Shuvo et al., 2024). However, these tools require endless integration, maintenance, and flexibility. Since they tend to contain static knowledge, they can become outdated and fail to provide comprehensive insights into the security landscape. (Adeniran et al., 2024).

Recent Artificial Intelligence (AI) advancements have proposed effective and innovative solutions for managing software and hardware vulnerabilities. Furthermore, these self-learning and proactive AI-driven models utilize advanced algorithms to detect vulnerabilities, significantly outperforming traditional static analysis tools (Ozturk et al., 2023). This enhancement diminishes the manual workload for security and safety engineers, encouraging them to focus on more strategic tasks. However, the industry remains cautious and selective in amalgamating AI techniques into security vulnerability management, emphasizing the significance of counteracting innovation with robust security and safety standards.

Fully automated vulnerability assessments leverage advanced AI techniques to make expert-like decisions without human intervention (Seas et al., 2024). Moreover, it is the most valuable method due to its potential for significant time and cost savings for users. However, the effectiveness of such automated techniques emphasizes the urgent need for a mature and robust framework comprised of AI code analysis and

Waheed, T., Marchetti, E. and Calabrò, A.
Vulnerability Mapping and Mitigation Through AI Code Analysis and Testing.
DOI: 10.5220/0013381500003896

In *Proceedings of the 13th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2025)*, pages 363-370
ISBN: 978-989-758-729-0; ISSN: 2184-4348

testing to address modern-day vulnerabilities.

This paper aims to provide a comprehensive methodology and framework for effective vulnerability mapping through AI code analysis and testing. The methodology, conceptualized in the paper into a framework, wants to provide an automatic vulnerability mapping and mitigation environment that can support developers in secure code generation even if they are not cybersecurity or testing experts. Specifically, the methodology outlines the following activities:

- creating a graphical representation of a source code to analyze its known vulnerabilities better;
- using the graphical representation to identify the part affected by the vulnerability and, if available, suggest and apply countermeasures;
- using the graphical representation to statically generate and execute security test cases, evaluate the impact of the applied mitigations, or discover possible (additional) security issues.
- exploiting AI support for vulnerability mapping, countermeasure application, and code generation

In this paper, Section 2 discusses related work and the current state-of-the-art. Section 3 discusses our methodology, and Our framework's vulnerability mapping and conceptualization are explained in Section 4 and Section 5, respectively. Section 6 concludes the paper.

2 RELATED WORK

In today's environment of disruptive technologies, continuous vulnerability assessments are critical for protecting the confidentiality, integrity, and availability of organizational and user data. Identifying key assets and regularly evaluating the vulnerabilities and possible data breaches can significantly reduce the risk of financial and reputation losses. Examples such as the SolarWinds (Alkhadra et al., 2021) and ransomware attacks (Olabim et al., 2024) highlighted the criticalities of cyber threats and demonstrated the weaknesses of traditional security measures. Adopting proactive vulnerability analysis strategic and monitoring activities could offer various benefits, mitigating significant cyber threats and supporting robust cybersecurity frameworks tailored to both technical and business needs. To this purpose, the rest of this section provides an overview of the current solutions for facing the vulnerability issues.

Among them, AI and deep learning models (Pooja et al., 2022) are popular for software vulnerability detection as they automate feature extraction. Word2Vec [1], a family of model architectures that can be used to learn word embeddings from large datasets, is the most commonly used feature vectorization technique in deep learning vulnerability detection models. At the same time, Bidirectional Long Short-Term Memory (BLSTM) is the most widely used algorithm. Graph neural network (GNN) based vulnerability detection models perform best when using Code Property Graphs (CPG) for feature representation and Word2Vec for feature vectorization, with the Gated Graph Sequence Neural Networks (GGNN) model outperforming other GNN architectures.

AutoVAS (Jeon and Kim, 2021) is a deep learning-based automated vulnerability analysis system. It represents source code as embedding vectors and achieves superior performance compared to other approaches, including detecting several zero-day vulnerabilities. While in (Bilgin et al., 2020) the authors demonstrated that a partial representation of the source code's Abstract Syntax Tree (AST) can still be effective for vulnerability prediction, even when the whole AST cannot be extracted or processed.

A new era of Explainable AI (XAI) has started (Rajapaksha et al., 2023), such as LIME, which allows the model to visually represent the identified vulnerable source code segments to help developers understand and fix the vulnerabilities.

The researchers (Nath et al., 2023) have integrated AI and blockchain-based systems to detect and prevent source code vulnerabilities during software development, particularly in a remote work-from-home scenario. Moreover, the proposed blockchain-based tamper-proof system focuses on trust and transparency, enabling traceability and non-repudiation.

The AI Vulnerability Database (AIVD) framework (Fazelnia et al., 2024) has been proposed to systematically and dynamically identify, evaluate, and mitigate AI-specific vulnerabilities. The researchers face challenges (Suneja et al., 2020) with real-world noisy data, and learning vulnerability-specific models is easier than a global model.

Rigorous efforts are needed to evaluate the performance (Durgapal and Kumar, 2024) of various machine learning models, including Logistic Regression, Random Forest, K-Nearest Neighbors, and Decision Tree, in detecting different types of software vulnerabilities. Moreover, combining machine learning techniques (Cotroneo et al., 2024) with fuzzing could lead to more advanced systems for detecting and mitigating software vulnerabilities, potentially improving the cybersecurity of software applications.

The existing solutions generally do not provide a comprehensive approach or are limited to analyzing

[1] https://www.tensorflow.org/text/tutorials/word2vec

some key aspects required for a robust vulnerability assessment framework. This motivates the work presented in the paper.

3 METHODOLOGY

As technology continues to advance, so do the challenges associated with software security and the effort required to face the sheer volume of known vulnerabilities that exist within software libraries and frameworks. The Common Vulnerabilities and Exposures (CVE) databases are an important source of information for developers. It identifies commonly experienced vulnerabilities, evaluates their impact, and determines the appropriate countermeasures. However, developers, especially not cybersecurity experts or the recently defined prosumers, may not be familiar with security testing or may not have the expertise to exploit the CVE features.

The methodology presented in this section supports the security of the developed code by leveraging modern methodologies that combine vulnerability analysis with innovative test generation approaches. Therefore, utilizing Control Flow Graph (CFG) representations (Wang et al., 2024) offers a clear and intuitive way to visualize software structure. It helps developers to identify vulnerabilities more effectively and to focus on the most critical part of the code.

Moreover, employing AI to assist with vulnerability mapping and mitigation provides immediate support to automatically detect vulnerabilities, suggest countermeasures, and even generate secure code, significantly easing security practices and empowering stakeholders without extensive cybersecurity training. Figure 1 schematises the proposed approach.

Based on the source code of the product that is going to be analyzed, two possible activities can be executed: analyzing the known vulnerabilities through the consultation of the Common Vulnerability and Exposures (CVE) database or graphically representing the code using CFG ((*CVE Analysis* and *CFG* in the figure, respectively).

Indeed, available CVE can be used to detect well-known cybersecurity issues. In this case, by querying the CVE database, it is possible to get information about existing vulnerabilities on the libraries included or imported in the source code. In parallel, the CFGof the code can be derived to get a manageable, more intuitive representation.

The CFG can be used for mapping the detected CVE and exactly localizing the branches or the library affected (*Mapping vulnerability* in Figure1). Consequently, for each of the considered CVEs, a reduced representation of the CFG is derived (*V.CFG slicing*). In this case, all the parts of the code not directly affected by the vulnerability are removed from the control flow graph (slicing of the CFV) to simplify the possible criticalities detection and resolution.

Additionally, if countermeasures are available for the mapped vulnerabilities, they are applied to the sliced code, (*Countermeasure Mitigation* in Figure1), to obtain an improved secure version(*Secure Code* Figure1). Otherwise, the sliced code is provided as input to a *Test generator* (*Test generator* in Figure1. It uses standard test strategies, like Path coverage or penetration testing, for security test case derivation.

The test cases are then statically executed (*Test Execution* in Figure 1, and the results collected and analyzed(*Verdict Analyzer* Figure1. In case of failures the AI facilities provides supporting solutions(*AI Mitigation* in Figure 1) for the generation of a new code version (*Secure code* in Figure 1.

A testing phase is executed in case even when code does not contain known vulnerabilities. In this case, its CFG (not its sliced versions) is used for test case generation, execution, and verdict collection.

In summary, the proposed methodology supports software vulnerability management and provides a solution for robust code production. It integrates advanced methodologies of AI, vulnerability detection, and testing to create a safer coding environment and support for building secure and reliable software.

4 VULNERABILITY MAPPING

As described in Section 3, two crucial activities are vulnerability mapping and countermeasures mitigation. In this section, more details about their role are provided. Specifically, as highlighted in Figure 2, vulnerability management is a persistent, proactive process to identify, assess, prioritize, and rectify security vulnerabilities in an organization. Moreover, the vulnerability management process involves the identification of various critical software vulnerabilities that potentially halt an organization's reputation and business continuity. Furthermore, it's crucial to understand the following vulnerabilities to enhance our vulnerability analysis and mitigation strategies:

Buffer Overflow: This vulnerability usually occurs when a program writes additional data to a buffer (a contiguous memory block) than its assigned limit. Moreover, the extra value or data can overwrite adjacent memory, leading to unpredictable behavior, crashes, or even allowing cyber-attackers to execute malicious code.

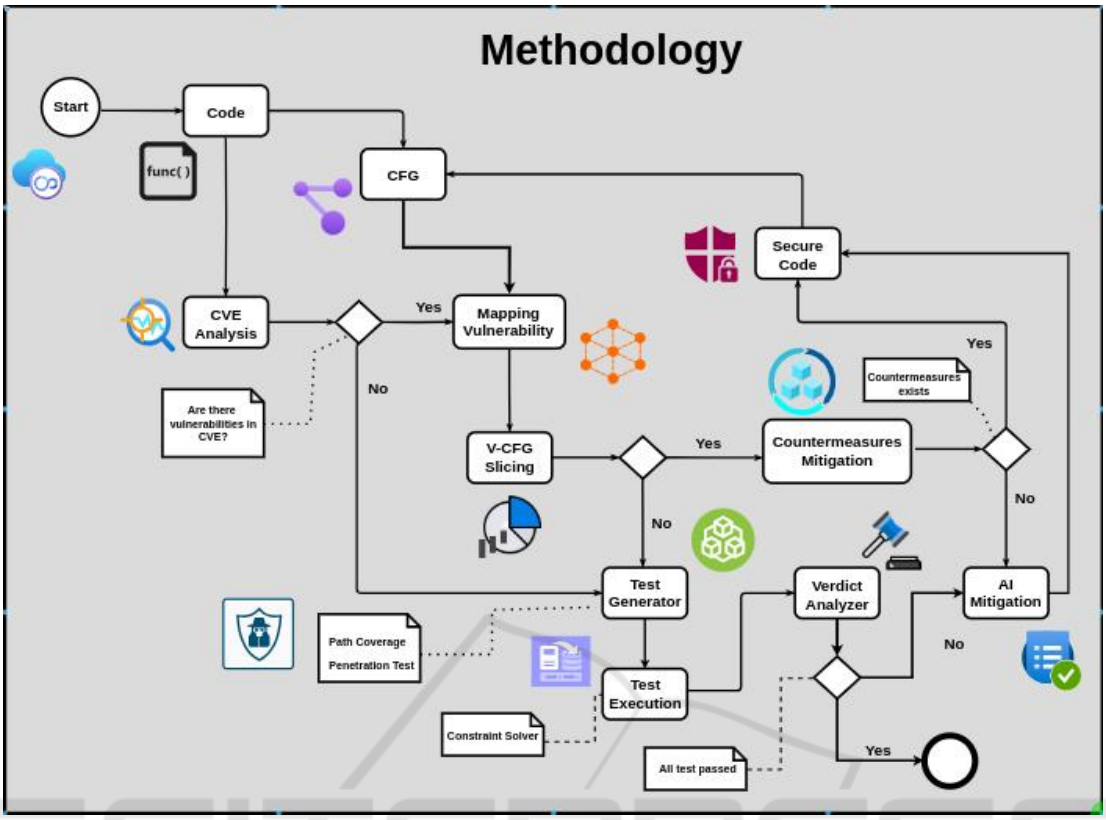

Figure 1: A graphical representation of the proposed Methodology.

Cross-Site Scripting (XSS): These vulnerabilities enable hackers to inject harmful scripts into web pages that abrupt users view to compromise sensitive user data, such as cookies or session tokens, or to ultimately acquire user credentials.

SQL Injection (SQLi): Attackers can insert malicious SQL queries that manipulate the database by exploiting weaknesses in an application's software. This can facilitate unauthorized access, data theft, or even complete database control, posing severe risks to confidential information.

Insecure Direct Object References (IDOR): Hackers can exploit weak points within the source code when applications inadvertently expose references to internal objects. By manipulating these references, they can gain unauthorized access to sensitive resources or confidential data.

Cross-Site Request Forgery (CSRF): This vulnerability manipulates legitimate users into executing unwanted actions on a web application, which can lead to unauthorized transactions or changes. By exploiting user trust, attackers can manipulate user interactions without their consent.

Path Traversal: This issue arises when user input is improperly used to construct file paths, allowing attackers to navigate beyond intended directories. By exploiting path traversal vulnerabilities, they can access restricted files and directories on the server, potentially jeopardizing sensitive data.

Invalidated Input: Many vulnerabilities occur due to the need for proper input validation. This negligence can lead to various attacks, including XSS and SQL injection, as faulty input may allow malicious actions to be executed by the software or system due to a lack of testing on the user side.

Misconfiguration: Software and systems are often deployed with default settings or insecure configurations, which poses significant risks. Moreover, the misconfigurations can expose sensitive information and grant unauthorized access to attackers, making them easier targets.

Insufficient Session Management and Authentication: Flaws in session management and user authentication processes can permit attackers to hijack user sessions or access accounts without proper credentials. Strengthening these controls is crucial for safeguarding user information.

Race Condition: This occurs when multiple processes or threads concurrently access shared re-

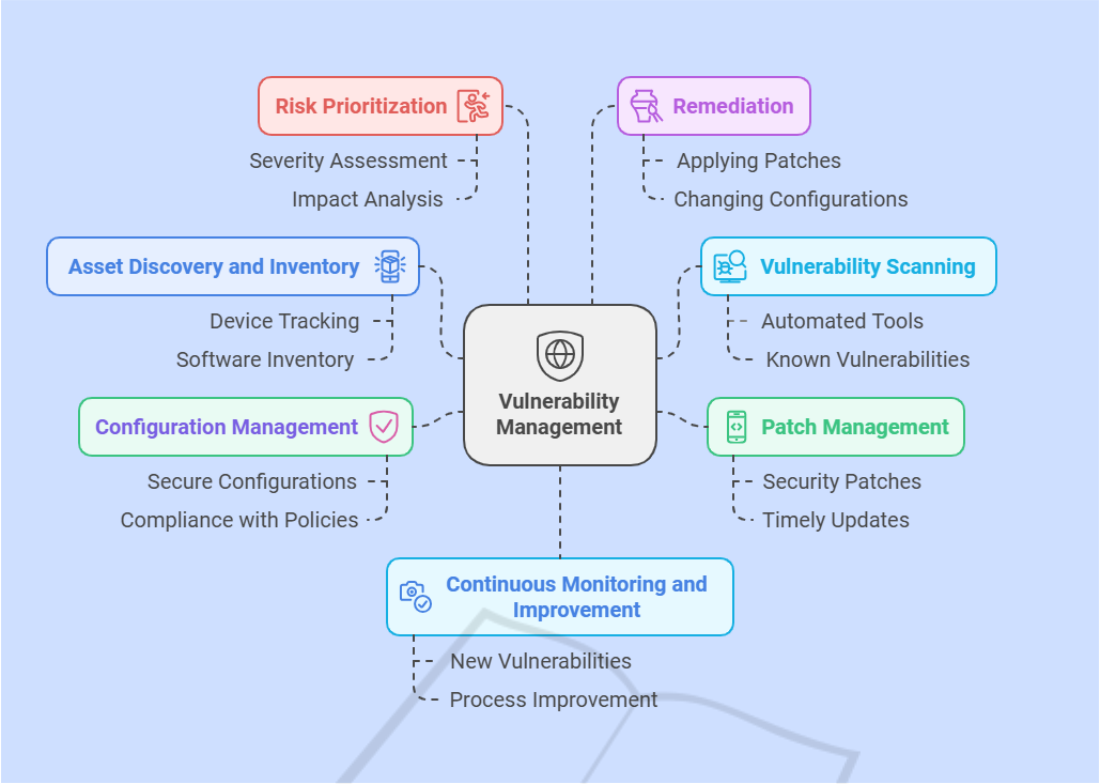

Figure 2: Vulnerability Management.

sources, leading to unpredictable behavior. Attackers can exploit these conditions to manipulate operations before resources are secured, potentially resulting in unauthorized actions.

Software Vulnerabilities for Legacy Systems: Older software often stops receiving security updates or patches, leaving it vulnerable to exploitation. These legacy systems can be attractive targets for attackers seeking to leverage known vulnerabilities. Denial of Service (DoS) While typically focused on network issues, software vulnerabilities may also enable attackers to overwhelm an application or service, rendering it unavailable to legitimate users. This can disrupt operations and harm user trust.

Addressing these vulnerabilities requires a strategic and systematic approach to vulnerability management, which encompasses the identification, assessment, prioritization, remediation, and ongoing monitoring of security issues. Regular software updates, comprehensive security audits, and automated vulnerability scanning tools are essential practices to proactively defend against potential cyber-attacks and leverage the overall security infrastructure.

5 CONCEPTUALIZATION OF FRAMEWORK

The methodology proposed in Section 3 conceptualized a possible solution to solve many of the issues mentioned in the related work (Section 2). In this section, we provide a preliminary implementation of the proposed methodology. In particular, for each of the activities presented in Figure 1, the following components have been considered:

Code: As schematized in Figure 1, the input of the proposed methodology is the code to be analyzed. Therefore, the precondition is the source code availability. Commonly used techniques for analyzing code that could be integrated into the proposed framework are GNN-based (Graph Neural Networks) models for C and C++ programs and deep learning (DL) models for open-source projects developed in various programming languages. However, as evidenced in the related works section, there is a lack of support for several programming languages beyond C/C++, and the necessity of a specific approach to vulnerability detection is still missing. The solution considered in this paper aims to leverage the current

limitation using a graphical representation of the code.

CFG: The source code is processed for deriving its Control Flow Graph (CFG) representation. It is essential to understand the flow of a program's execution and better highlight the critical parts to be modified or improved to reduce security risk. In this prototype version, the CFG is derived by integrating the open-source tool Code2flow[2]. Code2flow generates flowcharts from code, helping visualize logical processes and control flow. It supports multiple programming languages, simplifying complex code into easy-to-understand diagrams. The CFG enables focused vulnerability mapping and detection of critical paths and potential points of interest for security vulnerabilities.

CVE Analysis: As introduced in Section 3, main programming languages have a CVE repository tracking vulnerabilities and proposing security improvements. A CVE report refers to a specific vulnerability found in software or hardware. Each report includes a description of the vulnerability and its possible impact. CVE analysis is implemented by an AI-based analyzer that queries the CVE repository and lists the existing vulnerabilities. If no vulnerabilities are identified in the CVE database, the process continues seamlessly to the next phase, test generation.

Mapping Vulnerability: It uses the list of vulnerabilities provided by the CVE Analysis and links each detected vulnerability to their underlying causes on the CFG. This action is realized by exploiting the Code2flow tool. It relies on a root cause mapping for our CFG, which includes the following activities: i) Mapping nodes and edges, i.e., tracing the logical flow of execution (nodes) and transitions (edges) to identify if affected by the vulnerability; ii) Pinpointing faulty paths, i.e., highlighting loops, dead code, or unreachable nodes that may contribute to issues; iii) Tracing Dependencies, i.e., identifying root causes like invalid conditions, missing branches, or unintended control transfers impacting system behavior. The systematic mapping ensures vulnerability identification on the CFG and highlights where possible mitigation strategies could be applied.

V-CFG Slicing: The Mapping Vulnerability output is then used for slicing the CFG of the source code. In particular, the Code2flow tool extracts specific parts of the CFG that can contain a vulnerability. Indeed, reducing the CFG helps understand, debug, and analyze the code by focusing only on relevant portions of the code while ignoring the rest. Using the V-CFG slicing technique, each detected vulnerability provides multiple subgraphs (V-CFG) that allow a more in-depth examination of potential security risks. This activity can be implemented using either Machine Learning (ML) or Artificial Neural Networks (ANN) models. Several tools will be available in this prototype implementation, considering that some of them specialize in a single programming language. For example, Frama-C[3] is specialized in C code while Krakatau[4] can be used for Java. However, the currently available solutions still need improvement and manual intervention to achieve effective results. Such enhancements are particularly critical when it comes to finer levels of granularity, like examining individual code slices or tokens, where nuanced insights are essential for effective vulnerability detection and mitigation.

Countermeasures and Mitigation: The V-CFGs can apply the countermeasures suggested in the CVE and mitigate the identified vulnerability. The terms countermeasures and mitigation are interlinked as countermeasures are actions or strategies designed to detect, prevent, or reduce the impact of security breaches in case of cyber-attacks. Moreover, mitigation lowers the risk related to vulnerabilities. Typical strategies include recommending cryptography, Multi-Factor Authentication (MFA), and validating input to prevent vulnerabilities such as buffer flow and SQLi in our code to achieve secure code. In the prototype, the modern-day Large Language Models (LLMs) from Open-AI have been integrated into the process. It focuses on the vulnerable code segment and evaluates the applicability of countermeasures and mitigation strategies.

Secure Code: The suggestion provided by the Countermeasures and Mitigation can be used for improving the source code and correcting possible security threads or risks. In the current prototype version, this step relies on manual intervention from the users. However AI-based approach can be considered for future improvements.

Test Generator: In case their countermeasures and mitigation strategies are unavailable, the V-CFGs or the full CFG derived by the source code is used as input for a dedicated test process. This stage is essential for validating the security level of the analyzed code and highlighting possible unknown criticalities. The Test Generator may include var-

[2]https://code2flow.com/

[3]https://frama-c.com/

[4]https://soot-oss.github.io/soot/

ious standard testing strategies. However, in the current version, the path coverage is considered. Different tools can be integrated to realize the component depending on the original language of the source code. In this prototype, the Large Language Models (LLMs) from Open-AI are used for having language-independent test case results.

Test Execution: The identified set of test cases is then statically executed to achieve in-depth vulnerability analysis. In the prototype, we have integrated constraint solver and Gecode[5]. It is an open-source toolkit designed for creating constraint-based systems and applications. It offers a powerful constraint solver known for its state-of-the-art performance while being modular and extensible. This flexibility allows developers to tailor solutions to specific needs. It possesses testing capabilities for various applications that rely on complex constraint-solving capabilities to enhance vulnerability mapping, code analysis quality, and software security.

Verdict Analyzer: After the test cases are executed, a thorough analysis of the results concerning predefined benchmarks or criteria serves as the guideline for evaluation. The test cases are assessed during this analysis to determine whether it has met the required standards, leading to a verdict of pass or fail. Suppose the outcome reveals that all tests have passed successfully and an optimized level of vulnerability coverage has been achieved, meaning that potential security risks have been effectively identified and addressed. In that case, the entire process will end. However, suppose the analysis indicates unsatisfactory results or specific vulnerabilities still need to be addressed. In that case, further AI-driven mitigation is considered, as schematized in Figure 1.

AI Mitigation: This phase involves AI techniques to generate actionable recommendations and comprehensive mitigation strategies. These methods significantly improve the testing processes and enhance vulnerability coverage, guaranteeing that test cases are executed rigorously and effectively. Moreover, the tools integrated are Open-AI and Gemini; the framework facilitates a seamless integration that enables an iterative workflow. This dynamic methodology leverages software security and reliability standards, ensuring that each iteration leads to stronger protections against potential threats and achieving secure code.

[5]https://www.gecode.org/Gecode

6 CONCLUSIONS

This paper has highlighted the importance of vulnerability analysis as a proactive measure for identifying and mitigating cybersecurity threats within computing infrastructures. As traditional manual and assistive methods present significant limitations, adopting fully automated vulnerability assessments powered by advanced AI techniques is a crucial development.

The proposed methodology emphasizes the creation of a comprehensive framework that facilitates automated vulnerability mapping and mitigation. By leveraging graphical representations of source code, the methodology aims to enhance the efficiency of identifying vulnerabilities and implementing countermeasures. This enables a more accessible pathway for developers to generate secure code, regardless of their expertise level in cybersecurity.

Integrating AI-driven solutions streamlines the vulnerability assessment process and empowers security professionals to focus on higher-level strategic planning and response. However, as we embrace these technological advancements, it remains vital to prioritize the establishment of robust security frameworks for the application of AI in this domain, ensuring a balanced approach to innovation and safety.

Pursuing effective vulnerability management (as shown in Figure 2), through AI code analysis and testing is essential to safeguarding business operations and advancing security in a rapidly changing digital landscape. Future research should continue to refine these methodologies and explore further avenues for enhancing automated security solutions to stay ahead of emerging threats.

The proposed methodology and its preliminary conceptualization are the starting point for several improvements that could include: i) the use of dynamic A analysis and runtime monitoring in addition to static analysis through the CFG; this approach can help detect vulnerabilities that may only manifest under specific conditions, allowing for more comprehensive security testing. ii) use of automated remediation suggestions, based on best practices and prior knowledge and collaborative tools to allow developers to share insights, experiences, and solutions regarding vulnerability remediation, resolution strategies, and feedback on the effectiveness of implemented countermeasures. iii) make the AI support for vulnerability mapping and suggestions to work in collaboration with machine learning algorithms for better improving the vulnerabilities and remediation patterns and predicting potential vulnerabilities in similar code. iv) expanded the methodology to sup-

port various languages and frameworks by integrating language-specific analysis tools and libraries commonly used in the industry.

By incorporating these ideas, the proposed methodology can evolve into more robust, user-friendly, and practical support that identifies and mitigates vulnerabilities and actively supports developers in producing secure and resilient software.

ACKNOWLEDGEMENTS

This work was partially supported by the project RESTART (PE00000001), and the project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NextGenerationEU.

REFERENCES

Adeniran, T. C., Jimoh, R. G., Abah, E. U., Faruk, N., Alozie, E., and Imoize, A. L. (2024). Vulnerability assessment studies of existing knowledge-based authentication systems: A systematic review. *Sule Lamido University Journal of Science & Technology*, 8(1):34–61.

Alkhadra, R., Abuzaid, J., AlShammari, M., and Mohammad, N. (2021). Solar winds hack: In-depth analysis and countermeasures. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–7.

Bilgin, Z., Ersoy, M. A., Soykan, E. U., Tomur, E., Çomak, P., and Karaçay, L. (2020). Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684.

Cotroneo, D., Improta, C., Liguori, P., and Natella, R. (2024). Vulnerabilities in ai code generators: Exploring targeted data poisoning attacks. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pages 280–292.

Durgapal, H. and Kumar, D. (2024). Software vulnerabilities using artificial intelligence. In *2024 International Conference on Electrical Electronics and Computing Technologies (ICEECT)*, volume 1, pages 1–6. IEEE.

Fatima, A., Khan, T. A., Abdellatif, T. M., Zulfiqar, S., Asif, M., Safi, W., Hamadi, H. A., and Al-Kassem, A. H. (2023). Impact and research challenges of penetrating testing and vulnerability assessment on network threat. In *2023 International Conference on Business Analytics for Technology and Security (ICBATS)*, pages 1–8.

Fazelnia, M., Moshtari, S., and Mirakhorli, M. (2024). Establishing minimum elements for effective vulnerability management in ai software. *arXiv preprint arXiv:2411.11317*.

Janovsky, A., Jancar, J., Svenda, P., Chmielewski, Ł., Michalik, J., and Matyas, V. (2024). sec-certs: Examining the security certification practice for better vulnerability mitigation. *Computers & Security*, 143:103895.

Jeon, S. and Kim, H. K. (2021). Autovas: An automated vulnerability analysis system with a deep learning approach. *Comput. Secur.*, 106:102308.

Nath, P., Mushahary, J. R., Roy, U., Brahma, M., and Singh, P. K. (2023). Ai and blockchain-based source code vulnerability detection and prevention system for multiparty software development. *Computers and Electrical Engineering*, 106:108607.

Olabim, M., Greenfield, A., and Barlow, A. (2024). A differential privacy-based approach for mitigating data theft in ransomware attacks. *Authorea Preprints*.

Ozturk, O. S., Ekmekcioglu, E., Cetin, O., Arief, B., and Hernandez-Castro, J. (2023). New tricks to old codes: Can ai chatbots replace static code analysis tools? In *Proceedings of the 2023 European Interdisciplinary Cybersecurity Conference*, EICC '23, page 13–18, New York, NY, USA. Association for Computing Machinery.

Pooja, S., Chandrakala, C., and Raju, L. K. (2022). Developer's roadmap to design software vulnerability detection model using different ai approaches. *IEEE Access*, 10:75637–75656.

Rajapaksha, S., Senanayake, J., Kalutarage, H., and Al-Kadri, M. O. (2023). Enhancing security assurance in software development: Ai-based vulnerable code detection with static analysis. In *European Symposium on Research in Computer Security*, pages 341–356. Springer.

Seas, C., Fitzpatrick, G., Hamilton, J. A., and Carlisle, M. C. (2024). Automated vulnerability detection in source code using deep representation learning. In *2024 IEEE 14th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0484–0490. IEEE.

Shuvo, A. M., Zhang, T., Farahmandi, F., and Tehranipoor, M. (2024). Flat: Layout-aware and security property-assisted timing fault-injection attack assessment. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*.

Suneja, S., Zheng, Y., Zhuang, Y., Laredo, J., and Morari, A. (2020). Learning to map source code to software vulnerability using code-as-a-graph. *arXiv preprint arXiv:2006.08614*.

Waheed, T. and Marchetti, E. (2023). The impact of iot cybersecurity testing in the perspective of industry 5.0. In *International Conference on Web Information Systems and Technologies*.

Wang, P., Liu, S., Liu, A., and Jiang, W. (2024). Detecting security vulnerabilities with vulnerability nets. *Journal of Systems and Software*, 208:111902.