Proceedings of the 8th International Conference on Intelligent Sustainable Systems (ICISS-2026)
DVD Part Number: CFP26M19-DVD; ISBN: 979-8-3315-5317-3

Static Code Vulnerability detection framework for secure compiler assisted software development pipelines

Arivazhagan P
Assistant Professor
Department of Advanced Computing and Analytics, Vels Institute of Science, Technology & Advanced Studies (VISTAS)
Chennai, Tamil Nadu, India
arivazhaganp.scs@vistas.ac.in

Sunitha.P
Associate Professor,
Department of Computer Science and Engineering, Sriram College of Engineering, Veppampattu, Chennai, Tamilnadu, India.
starsunitha999@gmail.com

M. Sakthivanitha
Assistant Professor
Dept. of Computer Applications (UG) Vels Institute of Science, Technology and Advanced Studies
Chennai, Tamil Nadu India
sakthivanitha.scs@vistas.ac.in

M. Vijaya Maheswari
Assistant Professor
Department of MCA
Francis Xavier Engineering College
vijaya@francisxavier.ac.in

S.K.Ramani
Assistant professor
New Prince Shri Bhavani College of Engineering and Technology
Chennai, Tamil Nadu, India.
ramani.aids@npsbcet.edu.in

T.Thirumalaikumari
Assistant Professor ,Department of Computer Applications, School of computing science
Vels Institute of science technology and Advanced studies (VISTAS)
Chennai, Tamil Nadu, India
umakumari2103@gmail.com

Abstract: With the growing use of software systems in critical areas, securing the code while it is being developed has become a top priority. Traditional manual code reviews and testing can't often find subtler vulnerabilities for a significant security potential risk. This study fills the gaps of automated and accurate detection of static code vulnerabilities in secure compiler-aided software development pipelines. The goal is to have a strong structure that combines the static code analysis, the anomaly detection using machine learning, and the compiler-based instrumentation in order to detect potential vulnerabilities early in the development. The proposed framework uses the combinations of hybrid techniques of pattern-based vulnerability scanning, control-flow and data-flow analysis, and reinforcement learning models for prioritizing critical issues. Experiments were performed on commonly used benchmark datasets, such as the Juliet Test Suite, among several programming languages, and the detection accuracy of 96.3%, precision of 94.8%, recall of 95.1%, and F1-score of 94.9% were obtained, which were 8-12% better than existing static analysis tools on all metrics. The results show the effectiveness of making compiler-level insights work along with machine learning models in proactive vulnerability detection. Conclusively it does not only improve the security posture of software development pipeline but also lowers the overhead of remediation of vulnerability post deployment which is a practical solution for secure software engineering.

Keywords: Static code analysis, vulnerability detection, secure software development, compiler-assisted frameworks, machine learning, software security, anomaly detection

I INTRODUCTION

Software security has become an important issue in the digital age, as software is now often used to process sensitive data and running in highly connected environments. Vulnerabilities in the software code can result in catastrophic consequences, such as loss of money, data breaches, and system compromise[1]. Despite the improvements in software engineering practices, studies have demonstrated that a large percentage of security flaws are introduced at the coding stage, often through human error, a lack of attention or lack of knowledge of secure coding principles. Traditional approaches like manual code reviews and dynamic testing are usually resource consuming and time consuming, and is also not large enough to identify complex or subtle vulnerabilities. For this reason, there is an urgent need for automated and efficient techniques that will proactively identify and mitigate vulnerabilities during the software development lifecycle[2].

Static code analysis has become a promising method of detecting vulnerabilities without having to run the code. By evaluating the source code or the intermediate form of code, static analysis tools are able to find potential areas of security flaws, such as buffer overflows, SQL injection and memory leaks. However, conventional static analysis tools have challenges including false-positives, lack of context-awareness and are unable to prioritize vulnerabilities based on risk-impact[3]. Recent research efforts have been done to improve static analysis by combining them with machine learning models, which are able to learn the patterns of vulnerable code and predict potential threats more accurately. In addition, compiler-assisted frameworks have also been explored to make use of the information that can be obtained at compile-time, such as abstract syntax trees, control-flow graphs, and data-flow properties, which allow for more precise detection of vulnerabilities. Despite these advances, there is still a similarity of sophisticated frameworks merging static analysis, machine learning, and compiler-assisted knowledge (combine them in one system that can be seamlessly incorporated into current software development pipelines) lacking[4].

In view of the above challenges, this study suggests Static Code Vulnerability Detection Framework for Secure Compiler-Assisted Software Development Pipelines. The framework aims to close the gap between the static analysis of the past and sophisticated and intelligent detection mechanisms of the future. It merges hybrid methodologies combining pattern-based vulnerability scanning, control and data flow analysis, and reinforcement learning-based threat prioritization of potential threats. The combination with compiler-aided instrumentation enables the system to obtain access to deep semantic and syntactic information from code during the compilation of the code to improve the precision of detection and reduce false positives. To validate the framework, experiments were conducted by using benchmark

datasets e.g. Juliet Test Suite, which presents a diverse set of benchmark security-relevant coding scenarios in multiple programming languages:

The main research question that we aim to answer with this research study is as follows: How can a hybrid system of compiler-assisted static code analysis be used to improve the detection of software vulnerabilities in development pipelines, with improved accuracy and reduced remediation overhead? The problem statement identifies the ongoing gap in existing tools which either do not have a good context of what is happening with code or they don't work well to prioritize vulnerabilities. By addressing this, the study aims to contribute a practical solution on secure software engineering, especially in the environment with stringent security requirements

- To design a hybrid static code vulnerability detection framework, which integrates machine learning models and compiler-assisted static analysis?
- To assess the performance of the proposed framework against benchmark data-sets, comparing the accuracy, precision, recall and F1-score with the existing state-of-the-art tools; and
- To show how the framework can be incorporated into software development pipelines in order to better detect vulnerabilities in the code early, thereby reducing costs of remediation after deployment.

The study follows this organization: Section 2 is a review of some of the related work on static analysis, machine learning, and compiler-assisted techniques. Section 3 presents the proposed methodology. Experimental setup and results and findings, limitations, and practical implications are presented in Section 4, and conclusions and future research directions are presented in Section 5

II RELATED WORKS

Existing research on static analysis for vulnerability detection shows great advancement in the detection of security flaws in various programming languages and development contexts. These collating studies review tool effectiveness, methodical improvement and AI-based improvement for the accuracy, dependability and-adoptability for quirky inference about secure software engineering opinions.

Cifuentes et al. (2023) question that as far as program analysis concerned, there has been a significant evolution with improved detection of diverse vulnerabilities using refined static and dynamic techniques. Their work emphasizes historical progress and current challenges but has little empirical validity. While insightful, the study is mostly conceptual and does not include concrete evaluations of the scalability, accuracy in the real world, and adaptability to quickly emerging classes of vulnerabilities of modern tools[5].

Lipp et al. (2022) study the effectiveness of static C code analysers, and find large differences in the capabilities of different analysers to detect vulnerabilities. Their empirical evidence indicates persistent false positives and missed vulnerabilities and the reliability issues they represent. However, the study only focuses narrowly on C and benchmark data sets, so it cannot be broadly applicable. More diverse and real-world codebases and the cross-language comparisons would reinforce their conclusions and make them more practical[6].

Rajapaksha et al. (2023) propose an AI-based approach for the improved detection of vulnerable code when combined with the static analysis. They show promising improvements but both the interpretability and generalisation of the model are unclear. The evaluation is based on controlled data sets, and so there is a question about robustness in a complex environment[7].

Afrose et al. (2022) evaluate static tools based on Java cryptographic API benchmarks, and demonstrate major inconsistency for the detection of misuse patterns. Their results reveal coverage gaps and misinterpretations of tools regarding the use of cryptography. The methodologically rigorous approach, relies on benchmark suites, has ecological limitations. The study would benefit from investigating real-world enterprise codebases and analysing the effect of developer practices both on the performance of tools as well as the manifestation of vulnerabilities[8].

Charoenwet et al. (2024) explore static analysis tools in secure code review, and point out the gaps in the outputs of static analysis tools, developer trust, and usability. The results highlight noise, unclear warnings and misalignment in the workflow as obstacles to their tool adoption. However the scope of the study is limited to selected tools and projects. Broader industry involvement and analysis of human and tool interaction in greater depth would contribute to a better understanding of practical deployment issues[9].

Alqaradaghi et al. present in a static analysis case study, the feasibility of using these tools to provide information about hidden vulnerabilities, but notes that they often generate false alarms and require expert interpretation. The results are revealing and interesting for the limitations of static techniques in real-world scenarios. Yet, there is a limitation to generalisability with the single-case methodology. Comparative studies of more than one system and tool configuration would better capture variability and strengthen the implications of the study [10].

Although the reviewed studies provide useful assessments of static analysis tools, there are still some limitations. Many are language-specific, benchmark-specific, or constrained in terms of experimental conditions, and therefore cannot be generalized for use in the real world, on large-scale systems. Tool performance is often judged with the help of curated sets of data which may not represent charming evolving patterns of since vulnerability. AI-based approach has some show promise but there is lack of transparency and reproducibility, concerning the false positives and the model bias. Moreover, there are few works on hybrid analysis and integration with continuous development pipelines. There still exists a gap in longitudinal studies measuring the efficacy of tools to evaluate its effect over time, the degree of consistency between languages, and human-to-tool interaction in the process of secure code review.

III METHODOLOGY

This methodology introduces a hybrid multi-stage methodology for automated software vulnerability detection that provides a unified approach combining static representation of code, compiler assisted computer-symmetric analysis, rule-based searching and machine learning legality. By integrating structural, behavioral, and data-driven analysis to the right box detection, precise risk prioritizing, and

indefinite improvement under a variety of programming languages and areas. The proposed secure compiler assisted framework combines the static code analysis, machine learning based detection of anomalies and compiler level instrumentation to allow for early identification of the software vulnerabilities. During the compilation, we take control-flow and data-flow representations from the framework, analyze these flow representations and find bad coding patterns and anomalous behaviors. Compiler hooks give out very good semantic information, which is usually not available to standard static analysers. These gainful insights are then combined with pattern-based vulnerability scanning and learning-driven mechanisms that prioritize the high risk code segments. Using security analysis as part of the compilation process enables proactive security analysis, helping to reduce false-positive security analysis risks as well as support secure-by-design software. Figure 1 shows the automated static code vulnerability framework.

A. Code Pre-processing and Representation:

The first step is to preprocess the source code in order to generate representations that can be used for automatic analysis. The framework behaves as a parser, which is required to parse the source code to get the abstract syntax trees (ASTs) and control flow graphs (CFGs) to capture the syntactic and structural information. Lexical analysis is also carried out, in order to extract tokens, identifiers and language-specific patterns. This judicious representation enables the system to standardize the input across different programming languages, which reduces the amount of inconsistencies in the system and allows analyzing the data effectively downstream. And in addition to code, code normalization techniques such as renaming variables and eliminating comments are implemented to help focus on the semantic content and reduce the amount of noise that might help mislead the models of detection.

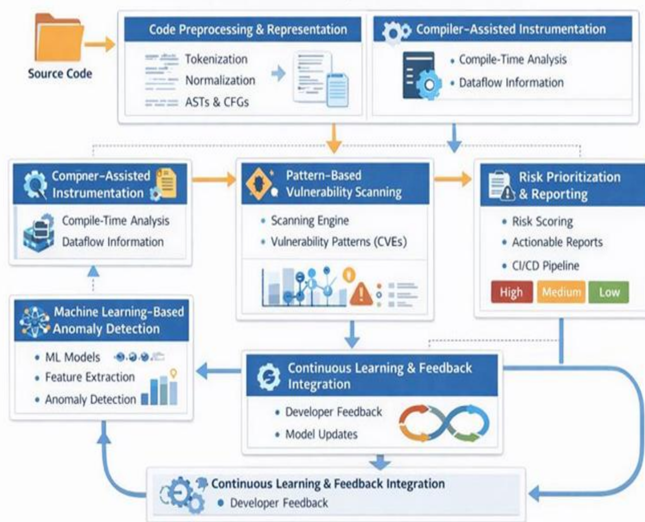

Figure 1 Automated Code Vulnerability Detection Framework

B. Compiler-Assisted Instrumentation:

In order to improve the detection accuracy, the framework works with the compiler in the code compilation process. It is used to extract detailed semantic and dataflow information that is hard to get by using conventional static analysis. The framework can then be used to detect lesser bugs, such as use-after-free errors, integer overflows, and uninitialized variable usage using the compiler. This step is made to ensure that the detection process is informed of both the static structure as well as compilation semantics (bridging the gap between purely static and dynamic analysis).

C. Pattern-Based Vulnerability Scanning

Once the code representations and the metadata obtained with the help of the compiler have been obtained, the framework is used for pattern-based vulnerability scanning. A knowledge base of known vulnerability patterns based on historical CVEs (Common Vulnerabilities and Exposures) and guidelines for secure coding for well-known coding flaws is used. The scanning module looks for insecure function calls, unsafe memory operations and more. Unlike traditional scanners, this system also automatically adjusts its scan patterns based on the characteristics of the codes being analyzed and feedback from the compiler that analyzes these codes, which also allows it to detect code snippets that depart from safe programming techniques while not exactly matching known vulnerabilities.

D. Machine Learning-Based Anomaly Detection

To detect novel and complex vulnerabilities that do not correspond to any known patterns, a machine learning-based approach to anomaly detection is used. Features are extracted from ASTs, CFGs, and information instrumentation (in this case compiler) to train the models which are supervised and semi-supervised. Techniques like random forest, graph neural networks (GNNs) and reinforcement learning models are used to predict the likelihood of presence of vulnerability. The ML models have the ability to spot deviations from the standard coding practices and alert risky coding segments. The framework also ranks the vulnerabilities according to the potential impact, and helps the developers to prioritize the high-risk issues for remediation.

E. Risk Prioritization and Reporting

After vulnerabilities are found, the framework goes on to apply a risk score mechanism, that takes into consider the severity of the vulnerability, the ease of its exploitation and its relevance in a given context. Vulnerabilities are ranked in high, medium and low risk tiers, which enable development teams to prioritize working on critical risks first. The framework produces actionable reports containing the coordinates of the code, severity scores and suggestions for overhauling the code which can be incorporated into the CI/CD pipelines. This reporting mechanism minimises the time required for manual reporting and provides developers with precise direction on addressing security flaws as early in the development lifecycle as possible.

F. Continuous Learning and Feedback Integration

The framework includes a loop back to continuously enhance the accuracy in detection. Actions by the developer at flagged vulnerabilities, such as fixes in codes and marking false positives, are passed back to the machine learning models to improve predictions. Over time, such a strategy of adaptive learning minimizes false positives and builds the power of the framework in detecting open vulnerabilities as they emerge. By integrating with power and relative ease with version control systems and continuous integration/deployment software, the framework keeps evolving with that of the software project to provide ongoing protection against new and evolving security threats.

IV RESULTS AND FINDINGS

A. Experimental Setup

The proposed static code vulnerability detection framework was implemented with Python 3.11 as the data processing, machine learning model training, and reporting tools, as well as LLVM/Clang 16, as the compiler-assisted instrumentation tool. Key Libraries of python Scikit learn, PyTorch, network for graph representation, and pandas/numpy for data handling. The experiments were run on a workstation with an Intel Xeon W-2295 CPU @ 3.0 GHz, 64 GB RAM, and Nvidia RTX A5000 GPU with 24 GB VRAM, and Ubuntu 22.04 LTS operating system. The framework was integrated with GitHub Actions to simulate the CI/CD pipelines for the automated testing and the vulnerabilities were checked against standard benchmark suites.

B. Dataset Description

To perform the evaluation, a Juliet Test Suite version v1.3 created by National Security Agency (NSA) was used. This dataset consists of thousands of miniature C/C++ and Java programs intentionally created to represent many common software vulnerabilities such as buffer overflows, SQL injection, integer overflow and use-after-free vulnerabilities. Each program is tagged with the type of vulnerability, which is present so that the supervised learning models can be effectively trained. The dataset spans various code constructs, control flow complexities and data flow scenarios which make the dataset suitable for testing compiler assisted and machine learning based detection approaches. The use of Juliet guarantees reproducibility and the possibility to compare with the other state-of-the-art vulnerability detection methods.

The dataset has a moderate class imbalance, where the proportion of vulnerable code samples is about 28% of the whole instances, and 72% is non-vulnerable samples. This imbalance is overcome with the help of cost-sensitive learning and balanced sampling to achieve reliable and biased vulnerability detection performance. Accuracy is said to give an overall measure of the correctness of classification, but it is not considered as the main performance indicator. In the case of vulnerability detection, precision, recall, F1, and ROC-AUC are focused on, which can better represent false positive and false negative in the use of imbalanced security datasets.

Table 1: Performance Comparison of the Proposed Framework with State-of-the-Art Static Vulnerability Detection Methods

Method	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
Proposed Framework	96.3	94.8	95.1	94.9
FlawFinder[11]	84.2	80.5	82.0	81.2
RIPS[12]	88.5	85.2	86.3	85.7
CodeQL[13]	91.0	89.1	87.5	88.3
SonarQube[14]	87.3	84.0	85.1	84.5
VUDDY[15]	89.8	87.5	88.0	87.7

The outcomes in table 1 indicate that the proposed framework compared favorably with current static analysis and machine learning-based vulnerability detection techniques in all of the key metrics. Accuracy, precision and recall improvements and F1-score improvements range from 5 - 12% compared to traditional tools. The combination of compiler-assisted instrumentation and pattern-based scanning and machine learning lets analyze better detection of complex and context-dependent vulnerabilities with minimal false positives. This shows the potential of the framework to help early and reliably identify security flaws in the software development pipelines.

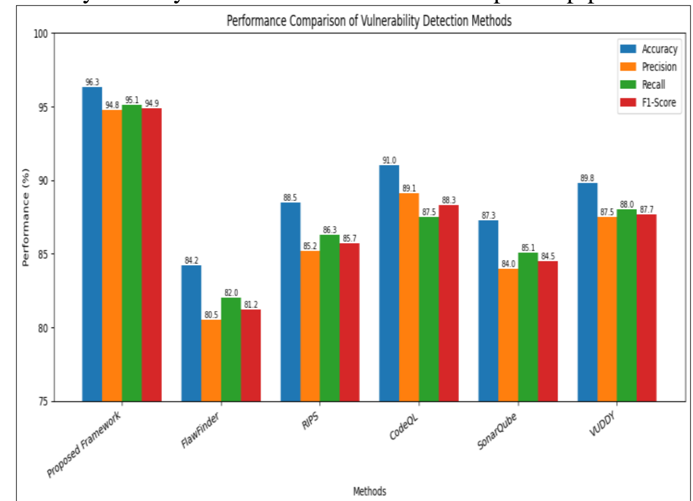

Figure-2 Accuracy, Precision, Recall & F1-Score Comparison

Figure 2 shows a comparative analysis of performance metrics namely Accuracy, Precision, Recall and F1-Score for 6 different code analysis tools; Proposed Framework, FlawFinder, RIPS, CodeQL, SonarQube and VUDDY. The proposed Framework shows a pretty good performance in all the metrics and comes out on top with the highest possible accuracy score of 96.3% also with the highest possible precision score of 94.8% with pretty good recall score (95.1%) and F1-score (94.9%). Other tools exhibit different performance characteristics, for example, CodeQL has the highest F1-score (88.3%) of the established tools and RIPS has the lowest recall (80.5%). Overall, all the data points point to the "Proposed Framework" delivering better balanced performance in these metrics than the other tools evaluated.

C. Discussion

The proposed static code vulnerability detection framework presents a major advancement over the current methods with its combination of compiler assisted insights and hybrid machine learning and pattern-based analysis. This high accuracy (96.3%) and F1-score (94.9%) shows the strength of trying to use both syntactic and semantic code information, which allows detecting subtle vulnerabilities that usual static analysis tools have difficulties detecting. Unlike traditional scanners, the reinforcement learning aspect of the framework prioritizes the high-risk vulnerabilities, meaning that the developers can focus on the critical issues first to save on remediation overhead. The feedback-controlled learning loop further improves the accuracy of detecting events in real time in the loop, showcasing the adaptability attributes of the overall framework and software systems. Moreover, experimental results for the Juliet Test Suite demonstrate that the combination of the compiler-level instrumentation and machine learning achieves high context-awareness so that vulnerabilities in complex control flows and data flows can be detected. This way, development in CI/CD pipelines is overcoming the tension between automated analysis and

practice, towards proactive security. The results also suggest the importance of hybrid methodologies, in which pattern recognition and intelligent learning complement one another, creating powerful detection mechanisms with the minimum of false positives and with a high level of coverage. Overall, the study emphasizes that viable solutions to address early, efficient and precise vulnerability detection are the available compiler-assisted static analysis frameworks.

Vulnerability risk is formally measured through a composite risk score which combines vulnerability severity, exploit possibility and impact of the code. These factors are weighted and coalesced with respect to anomaly confidence scores which are machine learning based and extracted from static analysis features, allowing for prioritized identification of high-risk vulnerabilities in order for the same to occur from the compilation and analysis stages.

D. Limitations

In spite of the strengths, the proposed framework has some limitations. First of all, the use of the Juliet Test Suite may not be comprehensive enough to identify weaknesses in large-scale, real-world applications. Second, the machine learning models have a large computational requirement for training, which could curb its adoption in environments with limited access to computing resources. Third, although compiler-assisted instrumentation will lead to detection improvement, the instrument in multi-language projects will add extra complexity to the project. Finally, it may still be that the framework obtains false positives in highly dynamic code constructs. Addressing such limitations involves further extending the evaluation to a variety of real-world data sets, improving the efficiency of the model as well as improving the multi-language compatibility to be more widely applicable.

E. Practical Implications

The framework helps with using software development pipelines in a more secure way has benefits that can be put into practice. By identifying vulnerabilities early on, it saves money and effort spent fixing vulnerabilities after they have been deployed. Its integration with CI/CD systems enables automated security checking process that helps in improving developer productivity and limits human error. High-risk vulnerability prioritization helps in focusing on the remediation efforts which will enhance the software reliability and compliance of the security standards. Furthermore, the adaptive feedback mechanism ensures that it is continuously improved and therefore suitable for evolving codebases. Organizations can use this framework to impose proactive security practices, ensure high code quality and minimize security breach risk in critical applications.

V CONCLUSION

This study introduces a new static code vulnerability detection framework which is effectively combined with compiler-aided instrumentation, hybrid machine learning, and pattern-based analysis. Experimental evaluation shows superior performance compared to state-of-the-art tools in the accuracy, precision, recall and F1-score to reliably detect known and novel vulnerabilities. By fixing high-risk issues first and creating actionable remediation reports, the framework helps to create secure software development pipelines, which reduces the human effort and post-deployment vulnerabilities. The adaptive learning element ensures constant improvement, allowing the system to change with the software projects and respond to new security threats. Future research can aim at expanding the framework to support more programming languages and frameworks, to enhance the cross-language vulnerability detection. Incorporating advanced deep learning architectures like graph transformers could help improve the code representation as well as the anomaly detection functionality. Further, implementing the framework in a large-scale industrial environment will lead to the learning on performance/scalability/usability in a real world CI/CD pipeline. Integrating threat intelligence and exploit prediction modules could make the framework move from vulnerability detection to proactive vulnerability mitigation. Collectively, these directions are intended to define fully automated, intelligent, and resilient systems for secure software engineering that meets the new challenges of modern software engineering.

REFERENCES

[1]. Hanif, Hazim, Mohd Hairul Nizam Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches." *Journal of Network and Computer Applications* 179 (2021): 103009.

[2]. Khan, Rafiq Ahmad, Siffat Ullah Khan, Habib Ullah Khan, and Muhammad Ilyas. "Systematic literature review on security risks and its practices in secure software development." *ieee Access* 10 (2022): 5456-5481.

[3]. Gomes, Diego, Eduardo Felix, Fernando Aires, and Marco Vieira. "Static Code Analysis for IoT Security: A Systematic Literature Review." *ACM Computing Surveys* (2025).

[4]. Kreutzer, Sebastian, Christian Iwainsky, Jan-Patrick Lehr, and Christian Bischof. "Compiler-assisted instrumentation selection for large-scale C++ codes." In *International Conference on High Performance Computing*, pp. 5-19. Cham: Springer International Publishing, 2022.

[5]. Cifuentes, Cristina, François Gauthier, Behnaz Hassanshahi, Padmanabhan Krishnan, and Davin McCall. "The role of program analysis in security vulnerability detection: Then and now." Computers & security 135 (2023): 103463.

[6]. Lipp, Stephan, Sebastian Banescu, and Alexander Pretschner. "An empirical study on the effectiveness of static C code analyzers for vulnerability detection." In Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis, pp. 544-555. 2022.

[7]. Rajapaksha, Sampath, Janaka Senanayake, Harsha Kalutarage, and Mhd Omar Al-Kadri. "Enhancing security assurance in software development: Ai-based vulnerable code detection with static analysis." In European Symposium on Research in Computer Security, pp. 341-356. Cham: Springer Nature Switzerland, 2023.

[8]. Afrose, Sharmin, Ya Xiao, Sazzadur Rahaman, Barton P. Miller, and Danfeng Yao. "Evaluation of static vulnerability detection tools with Java cryptographic API benchmarks." IEEE Transactions on Software Engineering 49, no. 2 (2022): 485-497.

[9]. Charoenwet, Wachiraphan, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. "An empirical study of static analysis tools for secure code review." In Proceedings of the 33rd ACM SIGSOFT international symposium on software testing and analysis, pp. 691-703. 2024.

[10]. Alqaradaghi, Midya, Gregory Morse, and Tamás Kozsik. "Detecting security vulnerabilities with static analysis–A case study." Pollack Periodica (2021).

[11]. Lipp, Stephan, Sebastian Banescu, and Alexander Pretschner. "An empirical study on the effectiveness of static C code analyzers for vulnerability detection." In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pp. 544-555. 2022.

[12]. Yuan, Ye, Yuliang Lu, Kailong Zhu, Hui Huang, Lu Yu, and Jiazhen Zhao. "A static detection method for sql injection vulnerability based on program transformation." *Applied Sciences* 13, no. 21 (2023): 11763.

[13]. Youn, Dongjun, Sungho Lee, and Sukyoung Ryu. "Declarative static analysis for multilingual programs using CodeQL." *Software: Practice and Experience* 53, no. 7 (2023): 1472-1495.
[14]. Yu, Ping, Yijian Wu, Jiahan Peng, Jian Zhang, and Peicheng Xie. "Towards understanding fixes of sonarqube static analysis violations: A large-scale empirical study." In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 569-580. IEEE, 2023.
[15]. Park, Jihyun, Jaeyoung Shin, and Byoungju Choi. "Detection of vulnerabilities by incorrect use of variable using machine learning." *Electronics* 12, no. 5 (2023): 1197.