Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

AI-Powered Code Security and Risk Depenedency Analyzer

Mr. Lochan Gowda M
Department of Computer Science and Engineering S J B Institute of Technology Bengaluru, Karnataka, India

Nisha H G, NithinKumar S, Pavan Kumar K S, Pandari
Department of Computer Science and Engineering
S J B Institute of Technology, Bengaluru, Karnataka, India

Abstract - **The effectiveness of the numerous security tools available today is limited by how they operate independently and by what they can access within a Developer Workflow System (DWS), or DWS Library. In this document, I will describe a new method of integrating several types of security analysis into a single integrated Security Analysis Framework (SAF) that utilizes Dependency Vulnerability Scanning, Static Code Inspection, Data Flow Tracking through Taint Analysis, Confidence Scoring, AI-based Summarization, Multiple Dependency Formats and AST-based & Taint Analysis to track the flow of untrusted data through multiple pathways and to sensitive endpoints to evaluate potential risk, as well as assign Confidence Scores to the individual findings so that you can better prioritize them and avoid generating false positives. The result of using the SAF is a front-end that is continuously updated in real time and can produce visual analytics and human-friendly summaries of detected vulnerabilities and triage of Security Issues.**

Index Terms - Artificial intelligence, dependency scanning, software security, static analysis, and taint tracking.

I. INTRODUCTION

The proliferation of third-party libraries in the creation of desktop/client/server applications has increased the number of different programming languages used in a single application, as well as the number of dependent programming libraries, which increases the risk of exposing these applications to security vulnerabilities [1]. Previous studies have demonstrated that legacy static analysis tools have higher false positive rates than other detection mechanisms, as well as limited capabilities to model data flow and limited knowledge of the semantics of the programming language being examined. Although using machine learning to automate detection of security vulnerabilities has provided improved performance in comparison to traditional methods, continued challenges exist regarding program representation and contextual awareness of code with respect to the source of untrusted data. Because of this, development of effective hybrid detection methods is essential [5].

Recent developments in the fields of artificial intelligence (AI) and natural language processing have enabled the creation of large language models and continue to grow in importance to the area of vulnerability detection. Large Language Models (LLMs), when given structured input and high-quality prompts, can semantically capture and understand a codebase, and can provide suggestions for automated remediation of vulnerabilities based on their understanding of the code [7]. However, the ability of LLMs to detect vulnerabilities depends on the availability and degree of structured ability of the input provided to the LLM, the quality of prompts generated, and how LLMs can integrate existing methods of program analysis. Methods for detecting vulnerabilities have also continued to be developed and refined to support two-dimensional (bimodal) taint analysis (i.e., multiple types of data), the ability to consider configuration, as well as hybrid scanning methods that can better follow the flow of untrusted data through complex systems when dependencies exist [12].

II. RELATED WORKS

In the past several years, there has been a rising interest in creating mechanisms to automatically find vulnerabilities in code, developing secure coding methods for programmers to use, and providing tools that will help programmers scan their applications for vulnerabilities. Researchers have conducted survey studies on the state of research and development in automated vulnerability detection [1]. These surveys report on the limitations and barriers researchers have experienced while conducting studies about how automated systems detect vulnerabilities. Some of the identified barriers include high false-positive rates, poor multi-language support, and insufficient guidance to help developers fix identified vulnerabilities [5]. As such, researchers have turned to the development of hybrid systems utilizing a combination of Machine-learning (ML) models along with Deep Program Analysis (DPA) and the ability to understand code semantically at a highly detailed level. Recently, researchers have been conducting experiments exploring

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

how Large Language Models (LLMs) can provide an automated way for researchers to analyze code and find vulnerabilities. While researchers have found empirical evidence that LLMs do help when trying to comprehend code's functionality, LLMs still require multiple forms of input (i.e., structured data) and several baseline inputs to operate effectively. Researchers have provided examples of how LProtector and PromSec are implemented to use structured prompting and multi-staged reasoning as a method to enhance LLMs' output reliability and safety within practical security workflows [9].

The evolution of taint-tracking, static analysis, and dependency analysis has led to the establishment of hybrid detection methods. In contrast to traditional taint analysis, Bimodal Taint Analysis incorporates both syntactic and semantic signals, resulting in greater accuracy for complex data flows than is achievable via single-mode techniques [12]. Configuration-aware strategies, as evidenced by ConfTainter, highlight the importance of providing a detailed representation of the configuration context surrounding an application's execution to alleviate most false positive reports [13]. As a result, dependency / supply chain security research demonstrates that the need exists for standardized identifiers and a reliable method for mapping dependencies and vulnerabilities across multiple languages (i.e., CPE / CVE reconciliation) to properly identify vulnerabilities [2], [4] and prioritize response based on the dependency intelligence provided [14].

III. OVERVIEW

The proposed system implements a layered architecture for detecting vulnerabilities. This system includes the use of LLMs, static code inspection, taint flow tracking, and dependency analysis to overcome the deficiencies of single-point methods. Single-point methods are susceptible to low accuracy rates and high false positive rates due to a reliance on isolated syntactic, semantic, or dependency analyses. The foundation of the framework is dependency analysis which provides a method for overcoming the increased complexity created by utilizing numerous third-party libraries by normalizing the various package names and versions of third-party libraries to a standard identifier. The framework combines traditional methods of conducting static code inspection with an LLM-based method of detecting vulnerabilities. This hybrid method utilizes both pattern-based and AST-based analyses to improve the semantic understanding of both the program being evaluated and the many different languages that the program may be written in. Taint flow analysis and LLM-generated summaries track how untrusted inputs flow through the control and data flows of the program. Taint flow analysis combined with LLM-generated summaries provide a means of producing human-readable remediation guidance. Bimodal, and configuration-aware, taints are used to identify complex paths of vulnerability and structured prompts improve both the efficiency of triaging and the common understanding of vulnerabilities across teams.

IV. ARCHITECTURE AND WORKFLOW

Python and React work together to create a RESTful API in addition to other technologies like WebSocket. This allows for a modular, layered approach to security analysis. Previous research has demonstrated that multi-layered methods that use a mix of syntactic, semantic, and machine learning inputs are better able to provide both accuracy and scalability when detecting vulnerabilities than single methods [1],[4].

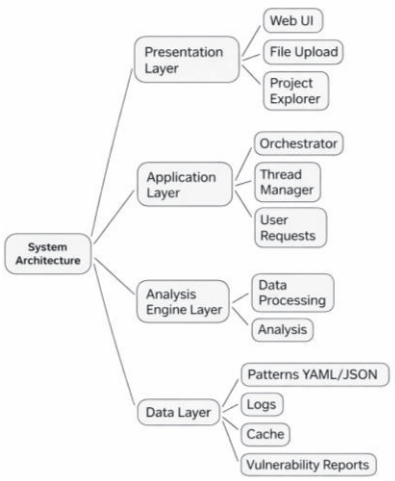

Fig.1 Architectural Design of AI-Powered Code Security & Dependency Risk Analyzer

The architecture defines how the system behaves, interacts, and processes data. It describes the internal workflow from input to analysis, scoring, and final reporting

A. Backend Architecture

A backend based on the Flask framework employs a modular design with individual Blueprints that encapsulate dependencies, Static Code Scanning, Taint Tracking, Scoring, and Summary to provide increased maintainability and verifiability [7]. To analyze scan requests, a parsing pipeline extracts Dependency information from supported Code Configuration Files and uses standard identifiers (e.g., CPE/CVE) to facilitate Accurate Vulnerability Matching across ecosystems and Reduce False Positive Risk [9]. For static analysis, rules-based pattern detection IS integrated with Abstract Syntax Tree (AST) traversal to Support detection of both Structural and Semantic Relationships that cannot be detected using Regular Expressions. Taint Tracking follows untrusted data propagation and uses bimodal and configuration-aware techniques to accomplish this;

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

confidence scores are based upon heuristics of sanitization presence and structural similarity, and scalability is accomplished via Parallel Execution mechanisms using Process.

B. Frontend Architecture

The frontend is a single-page application built in React that provides a modular interface to support easy upload of files, configuration of scan settings, visualization of the scanning process, and exploration of results to facilitate efficient triage and assist developers' understanding [5], [12]. The communication between the frontend and backend is managed through an Axios service layer, and the display of scan results adheres to best practices for multi-tiered vulnerability visualization via dynamic charts and tables [3], [6]. The provision of real-time progress updates on the scan is accomplished with the Server-Sent Events technology, which provides ongoing feedback to help build users' confidence when performing scans that may involve long processing times [5], [9].

C. End-to-End Workflow

After a user uploads a project and/or starts a scan (via Command Line Interface), the backend will extract and normalize all the dependencies, look up those dependencies against multiple vulnerability databases (as recommended in cross-language dependency analysis research [7], [10]), and create a single dataset (Database). While the dependencies are being processed, the source files will go through hybrid static analysis (which includes regex check, abstract syntax tree validation, and data flow tracking) - all of which are methods that have been shown to provide better results than just using one method [11], [12]. As soon as all the analysis results have been aggregated and scored, they trigger a large language model (LLM) to provide a structured and high-level summary of risk. Based on previous research, it has been established that using contextual-rich prompts for LLMs increases the reliability of LLMs for security workflows and reduces the likelihood of LLM hallucinations [5]. Finally, as a unified dashboard, these outputs represent all known dependency vulnerabilities, taint traces (which represent how data moves through the execution of an application), risk scores (for each true positive), and insights generated by LLMs regarding those risks. Together, this unified dashboard illustrates the implementation of modern multi-layered security practices within a single framework, producing actionable and user-friendly security intelligence [1].

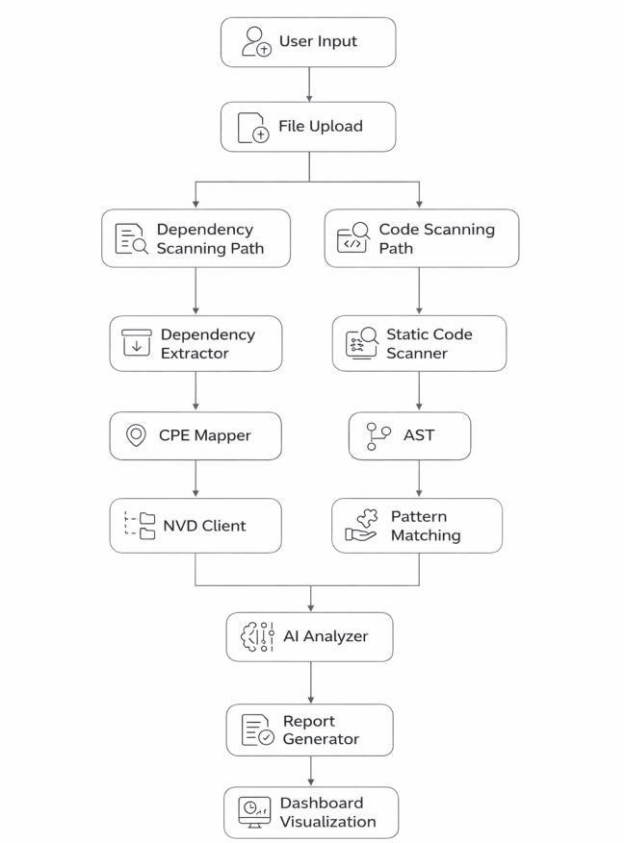

Fig.2 Design Flow of AI-Powered Code Security & Dependency Risk Analyzer

V. IMPLEMENTATION

This research paper describes how the integration of machine learning technologies into a multi-tiered software vulnerability detection (SVDT) system increases both accuracy and effectiveness. The SVDT Framework consists of five primary components, including Dependency Extraction, Static Code Analysis, Taint Analysis (referring to the tracing of application data from one point to another), Confidence Scoring, and Large Language Model (LLM) Summarization. Based on the survey research [1] of previous years that examined the ways of conducting vulnerability detection, the goal of this new hybrid architecture is to reduce false positive results by maximizing analytical precision throughout the system's various modules. In addition, the modular design of the SVDT Framework facilitates concurrent processing and complies with the current trend for developing scalable and producing high-quality reports of vulnerability analyses and detection [5].

A. Dependency Extraction and Normalization

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

To gather application metadata, dependency extraction operates on many distribution files (e.g., requirements.txt, package.json, pom.xml). The nature of software ecosystems creates differences in the way that application metadata is defined and represented; therefore, they need to be transformed and standardized [7], [8]. The identifiers extracted from dependency files are transformed into a common format through vendor normalization and a combination of internal heuristics, which ensures accurate match resolution for vulnerability purposes [9], [10]. The metadata that is normalized is then cached through MD5 or similar hashing mechanisms, which avoids unnecessary repeat searches for vulnerabilities and increases efficiency in the analysis process [11].

B. Static Analysis Engine

The hybrid model of the static analysis engine is used to minimize false positives resulting from purely syntactic analysis [12], [13]. To achieve this, the static analysis engine combines regex-based pattern matching with AST Traversal to eliminate falsely detected patterns. The use of configurable vulnerability patterns across languages and the use of AST node context has enabled the static analysis engine to capture both structural and semantic characteristics of code. By combining the use of syntactic signals and semantic signals for detecting patterns, the static analysis engine has improved its detection accuracy [5],[14].

C. Taint Flow Tracking

Taint analysis identifies vulnerabilities that occur as a result of the flow and propagation of data by attaching tags to these variables. Tags are Attached to the variables when they're created by an untrusted source, and then they are tracked throughout the control path and data path. Taint analysis is based on bimodal principles and combines syntactic signals that indicate control flow (e.g., branches, loops) with semantic signals that indicate data flow (e.g., functions that use the tag) to identify hidden propagation paths [15]. Techniques that are aware of configuration settings (e.g., decoupling, VDS) such as those provided by ConfTainter can further model how variables interact with one another over configuration states and function boundaries [16]. In this manner, simplified call paths are used to determine if tainted data can reach sensitive sinks (e.g., through invoking subprocesses or conducting database operations) without being properly sanitized.

D. Confidence Scoring Mechanism

To minimize False Positives and Prioritize Vulnerabilities, we created Context-Aware Confidence Scores for Realized Vulnerabilities by incorporating supporting factors such as: Sanitization, Similar Structure, and File Level Information [4].

E. Large Language Model–Based Summarization

To improve usability, an LLM generates human-friendly and concise summaries of multi-layered security analytics results. Through structured prompts using taint paths, dependencies, level of confidence, and relevant metadata, the generated output is enhanced in reliability and usefulness. Prior findings support that LLMs are best used with traditional methods of software program analysis [1]. The summarization module also provides actionable guidance to mitigate risk depending on users' level of security expertise.

VI. RESULTS AND DISCUSSION

By evaluating small, medium, and large software projects, there is strong evidence to support that combining dependency scanning with static analysis, taint tracking, and confidence scoring is a better solution than using single-layer tools for vulnerability detection, in alignment with earlier research on hybrid software security pipeline methods [1], [2]. In this context, static analysis was able to successfully find structural vulnerabilities that pattern-based scanners had missed, demonstrating the advantages of using semantic and AST practices [3]. Taint tracking identified high-risk paths of untrusted inputs to sensitive operations, confirming prior research regarding taint-based approaches for identifying vulnerabilities [4].

Table 1: Results Summary of the Multi-Layered System

Aspect	Key Findings
Hybrid Analysis	Higher coverage
Static Analysis	AST-based detection
Taint Tracking	Multi-hop flows
Scalability	Parallel execution
LLM Interpretation	Faster triage

Using modular architecture and parallel execution allows the system to strongly scale, making it suited to efficiently analyze large repositories and support deployment in a real-world CI/CD (Continuous Integration / Continuous Deployment) workflow. The system implements real-time streaming of progress and catching decreases analysis time and redundant vulnerability checks. The use of summary generation from LLMs (Large Language Models) helps to provide simple and concise summaries of vulnerabilities and remediation options to reduce triage time and to help view the LLM as an assisting tool rather than a full standalone security scanning tool. The data indicate that using a multi-layered and stacked approach provides a significant increase in both accuracy and usability compared to traditional one-layered techniques, consistent

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

with the prior studies of integrated vulnerability detection systems [3], [6]. The system provides more accurate and developer-centric insights into security than using any of its constituent elements alone by using static analysis, taint tracking, intelligence regarding dependencies, and LLM-based reasoning; as indicated in recent studies [5], [8], future uses of the system might provide more support to additional languages by supporting multilingual abstract syntax trees (ASTs) and in perhaps providing greater reliance on various graph-theoretic approaches to reasoning than might be available today.

VII. CONCLUSION AND FUTURE WORK

The current work has presented a multi-layer vulnerability detection system. The combination of multiple techniques of vulnerability detection is accomplished by combining dependency scanning, static code analysis, taint flow analysis, confidence scoring, and supported by large language model (LLM) enhanced summarization. This study has shown that using these multiple layers of detection provides greater accuracy and broader information about software security than using traditional single-method techniques, confirming the conclusion of several other studies [1], [2] that hybrid detection systems are superior to either isolated static or learning-based tools. The AST-based scanning and taint flow tracing of the current system found both structural and data flow vulnerabilities not typically recognized by a purely pattern-based tool, and LLM-assisted developer comprehension, and reduced time required for triaging, provides additional supporting evidence of the advantages offered by AI-assisted security workflows [3], [4].

Going forward, the potential to improve the functionality of the system through future enhancements is great. Modern analysis research [2] indicates that if AST support was expanded to other programming languages and if graph-based representations of code were used, the capability for semantic reasoning and multi-hop detection of vulnerabilities could be increased. Also, by improving LLM alignment and designing better prompts around LLMs, the amount of hallucination seen when producing remediation guidance will decrease, while also increasing the clarity of the guidance generated. Finally, by implementing support for incremental scanning and the ability to track historical vulnerability information, the system could be placed within continuous integration and delivery frameworks much more easily. Overall, this system has demonstrated that using both program analysis techniques and AI approaches to support reasoning about program analysis is a viable and effective way to support future research and practical deployment of this type of technology across multiple different software environments.

REFERENCES

[1]. Abid, K. Ren, H. Liu, and Y. Zhang, “Survey on security and privacy considerations of large language models,” *arXiv preprint*, 2024.

[2]. Y. Yao, K. Ren, and H. Liu, “Survey on the security and privacy of large language models: The good, the bad, and the ugly,” *arXiv preprint*, 2024.

[3]. Z. Sheng, T. Zhang, Y. Li, S. Xu, and Z. Qin, “Survey on software security through the usage of large language models and vulnerability detection techniques,” *arXiv preprint*, 2025.

[4]. M. S. H. Shaon, “Modern software vulnerability detection approaches: A survey of machine learning, deep learning, and large language models,” *Electronics*, MDPI, 2025.

[5]. J. E. Ameh, A. Otebolaku, A. Shenfield, A. Ikpehai, and D. Sule, “Machine learning–based approaches for software vulnerability detection: A systematic review,” *ResearchGate preprint*, 2025.

[6]. Y. Yao, Y. Zhang, and G. Shen, “Survey on software vulnerability analysis using programming language and program representation landscapes,” *arXiv preprint*, 2023.

[7]. C. Fang *et al.*, “Large language models for code analysis: Do LLMs really do their job?” in *Proc. USENIX Security Symposium*, Washington, DC, USA, 2024.

[8]. T. Wang *et al.*, “ConfTainter: Static taint analysis for configuration options,” in *Proc. IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, 2023.

[9]. Y. W. Chow, I. A. Ahmed, T. B. Kothari, and S. Debray, “Beware of the unexpected: Bimodal taint analysis,” in *Proc. USENIX Security Symposium*, 2023.

[10]. L. Li, Z. Yang, X. Chen, and Q. Zheng, “AI-powered vulnerability detection toward secure source code development,” *arXiv preprint*, 2024.

[11]. L. Li, Z. Yang, X. Chen, and W. Xu, “LProtector: A vulnerability detection system powered by large language models,” *arXiv preprint*, 2024.

[12]. R. T. N. Silva, M. D. V. Ribeiro, and E. C. Santos, “PromSec: Prompt optimization for the secure generation of functional source code using large language models,” *arXiv preprint*, 2024.

[13]. J. Sharma *et al.*, “A survey on machine learning techniques for source code analysis,” *arXiv preprint*, 2021.

[14]. Chafjiri *et al.*, “Systematic review of vulnerability detection using machine learning–based fuzz testing,” *Computers & Security*, 2024.

[15]. X. Zhu *et al.*, “Exploring software security and large language models: Present and future directions,” *arXiv preprint*, 2025.

[16]. N. O. Jaffal, “Cybersecurity applications of large language models: AI-generated vulnerabilities and defense strategies,” *Cybersecurity*, MDPI, 2025.

[17]. I. L. V. Silva *et al.*, “Probabilistic time integration for semi-explicit probabilistic differential algebraic equations,” *arXiv preprint*, 2022.

[18]. A. D. Smith *et al.*, “Joint mixed-effects model methods for causal inference from clustered network-based observations,” *arXiv preprint*, 2022.

[19]. Y. K. Leong *et al.*, “High-gain unidirectional dipole antenna with wide bandwidth for next-generation WLAN,” *IEEE Antennas and Wireless Propagation Letters*, 2023.

[20]. R. S. Almeida *et al.*, “Elastic scattering of supernova neutrinos and electrons via xenon,” *arXiv preprint*, 2024.