VULNDET: A Distributed System for Detecting Vulnerabilities in Project Dependencies

Abhinav Pandey
School of Computing
Amrita Vishwa Vidyapeetham
Kollam, India
meronaamabhinav@gmail.com

R.V. Rajagopalan
School of Computing
Amrita Vishwa Vidyapeetham
Kollam, India
rajagopalan602@gmail.com

Aadarsh T Rajeev
School of Computing
Amrita Vishwa Vidyapeetham
Kollam, India
aadarsht087@gmail.com

Abhinand N
Cyber Security Department
Amrita Vishwa Vidyapeetham
Kollam, India
abhinandn@am.amrita.edu

Parvathy R
School of Computing
Amrita Vishwa Vidyapeetham
Kollam, India
parvathyr@am.amrita.edu

***Abstract*— The widespread use of open-source software (OSS) introduces significant risks from insecure third-party dependencies. Existing tools for vulnerability detection are often platform-specific and lack centralized control. This paper presents VULNDET, a distributed, platform-agnostic system for detecting and remediating vulnerabilities in OSS project dependencies. Designed with extensibility, it supports Node.js and Python environments and employs language-specific modules for scalable detection. A key innovation is an automated patching mechanism that combines Docker-based sandboxing with lightweight LLM to generate and validate patches. VULNDET achieves a higher detection rate on a custom CVE-injected dataset compared with industry-standard tools. The system enables proactive, automated, and cross-platform OSS vulnerability management.**

***Index Terms*—Common Vulnerabilities and Exposure (CVE), Vulnerability detection, dependency security, open-source software (OSS), Node.js, Python, MongoDB, Distributed systems**

I. Introduction

Open-source software (OSS) has accelerated development cycles and reduced costs, but it also introduces security risks through vulnerable third-party dependencies. Unpatched flaws in these libraries can lead to data breaches and system instability, making dependency management a critical concern. These concerns are especially relevant in the Indian context, where a large number of educational and governmental institutions rely on OSS without sufficient mechanisms for formal vulnerability handling, exposing critical systems to known exploits [19].

Handling these dependencies can be a complicated process since different open-source libraries differ in quality and maintenance levels. Despite the availability of many tools, these mostly cater to specific platforms which often leads to vendor locks and diminished overall efficiency as far as comprehensive automated detection and centralized monitoring are concerned.

In contrast, the proposed system is a pure mechanism that does not work on any particular platform. It can be utilized in different settings without being constrained to any particular technological framework. Such an approach will give complete control over configuration, data management, and monitoring processes to the users. The system would automatically scan through third-party dependencies to identify security problems along with a critical CVE ID and suggested fix.

A primary database is used to store all vulnerability information to enable system administrators and security teams to monitor all the vulnerabilities and work on them as needed. This kind of setup is proactive in aiding them in identifying possible risk factors. Data is replicated from various devices simultaneously under a master-slave configuration making it convenient to scale up the system whenever necessary. Moreover, there are authentication log-ins and frequent verifications in place to help maintain the integrity and reliability of the information.

The paper also proposes an automatic patching mechanism. We utilize the use of an isolated Docker environment to patch the vulnerabilities and check the stability of the system. This method even makes use of LLM to generate build files whenever necessary. We propose a bigger discussion for the stability of such an automated module. This novel approach would require a lot of testing and improvements before confirming that it is stable. This module if implemented properly would solve a lot of problems existing in this domain.

This paper dives deep into the system's architecture and its scalable architecture and how well it can handle dynamic and complex workflows and setups. While this project focuses on Node.js and Python environments, the same approach can easily be extended for other languages and frameworks.

II. Related Work

Open-source dependencies are one of the significant security concerns, especially in large-scale and widely used software, where third-party libraries are crucial but pose a huge risk. Several approaches have been developed to detect and manage such vulnerabilities. This section reviews some relevant studies and tools, each contributing insights that inform the design and functionality of our proposed vulnerability detection system.

VulPecker is an automated vulnerability detection system using code similarity analysis to identify vulnerabilities in software [1]. It works by defining features of vulnerability patches and applying code-similarity algorithms to detect similar vulnerabilities in different copies of the software. Although VulPecker concentrates on C/C++ projects, the scalable similarity-based detection concept remains a basis for detecting repeated vulnerabilities that may appear in software dependencies across open-source ecosystems. Our system addresses this by leveraging native ecosystem tools and Language-specific

modules for better adaptability and maintainability across different programming environments.

Another critical research article explores the integration of Common Platform Enumeration (CPE) and Common Vulnerabilities and Exposures (CVE) datasets into better matching software packages to known vulnerabilities [2]. Their method emphasizes classification accuracy but remains passive—requiring manual correlation between datasets. In contrast, VULNDET automates this process using centralized logging and real-time updates from audited scans, enabling actionable insights without manual triage.

One study classifies the vulnerability detection tools based on their analysis method, namely static, dynamic, or hybrid, and tests how effective such tools are at detecting vulnerabilities commonly found in Node.js projects [3]. This research is especially relevant to our system because it puts a lot of emphasis on targeted, environment-specific scanning, which our system does by using npm audit scans specific to Node.js dependency security.

When researchers examined the incidence of vulnerabilities in npm dependencies, they found that many apps are susceptible due to outdated dependencies [4]. This research illustrates a common security issue in dependency management: programs that do not update packages on time expose themselves to known vulnerabilities.

Emerging approaches also explore AI-based techniques for vulnerability detection in more specialized domains. A recent study by A. M., S. R., and S. G. focuses on smart contract security using Graph Neural Networks (GNNs) to identify vulnerabilities like reentrancy, overflow, and access control flaws in Ethereum-based contracts [21]. Their work proposes two GNN-based models—one optimized for accuracy, and the other for cost-sensitive detection, prioritizing high-impact vulnerabilities. While the overall accuracy of the standard model was higher (96%), the cost-sensitive model, though slightly lower in accuracy (94%), achieved significantly better outcomes in reducing financial losses from undetected critical flaws. Similarly, Chandran et al. proposed a classification model for detecting Advanced Persistent Threats (APTs) [22]. Both works emphasize context-aware, automated detection, aligning with VULNDET's design goals. While VULNDET does not currently use machine learning, these studies point to future enhancements in intelligent, impact-driven vulnerability prioritization.

Our system's centralised database and continuous monitoring provide regular, automated vulnerability checks, which help to reduce this risk. GitHub's Dependabot tool offers a popular solution for dependency monitoring, automatically alerting users to vulnerabilities in their software dependencies and suggesting updates [5]. VULNDET decouples this constraint through a distributed master-slave model, making it suitable for heterogeneous infrastructures, including private networks or air-gapped systems.

To address dependency security in Python projects, Ochrona offers a free vulnerability detection tool that integrates with the Python ecosystem by scanning packages against an updated vulnerability database [6]. Ochrona's targeted support for Python dependencies informs the Python-specific component of our detection system, enabling it to identify security risks in Python environments and complementing the Node.js capabilities of npm audit.

A large-scale study [20] analyzed 32,164 vulnerabilities across 11,353 systems within 20 network labs of a global enterprise over three months. Their findings underscore the magnitude and complexity of vulnerability landscapes in distributed environments and highlight the urgent need for automated, scalable approaches to vulnerability detection and management. VULNDET addresses this gap by offering decentralized scanning with centralized reporting, making it suitable for similarly large and heterogeneous infrastructures.

III. Implementation

The architecture, components, and workflows of the vulnerability detection framework are designed to support scalable and automated tracking across multiple devices, with initial implementation targeting Node.js and Python dependencies. The system is structured into two core layers: General Setup, which performs device-specific analysis and detection, and Server-Level Architecture, which manages distributed monitoring and centralized processing. While currently focused on Node.js, Python, Java and Go, the framework is built to be extensible, allowing seamless integration of additional languages and systems such as Linux in future expansions.

A. Local Scanning Setup

The local scanning setup has been designed mainly to identify and analyze Node.js and Python projects on individual devices. This phase includes features such as dependency extraction, vulnerability detection, and secure data logging. The local scanning leverages the different Language Adaptation Modules (LAM) to perform language-specific dependency extraction and vulnerability assessment in individual devices.

The Language Adaptation Modules (LAM) that are in use provide a modular and extensible framework for integrating these language-specific capabilities into the vulnerability detection system. Each of the LAMs are tailored to handle the unique dependency extraction and vulnerability assessment requirements that are specific to a programming language. For example, by parsing package.json files and utilizing npm audit, the LAM-Node recognises Node.js projects, while LAM-Py, on the other hand, handles Python projects through method like virtual environment detection and scanning with tool such as Ochrona. The framework is designed for scalability, with planned extensions such as LAM-Java for Maven or Gradle-based Java projects and LAM-Go for Go modules, ensuring seamless support for additional ecosystems as per the system's future updates.

1) LAM-Node:

Node.js projects are identified by scanning directories for the presence of a `package.json` file, which is the primary configuration file for Node.js applications and includes critical information about the project's dependencies. The detection script performs a structured directory search, where it validates each `package.json` file by ensuring the absence of a node modules directory in any parent directory, as previously demonstrated by Alfadel et al. [4]. This validation stage sets apart primary Node.js projects from nested dependencies under node modules folders, which could hold packages unrelated to the main project.

After a primary Node.js project is verified, it is included in a project list to be further processed. For each valid project, the system interprets the package.json file to obtain a list of dependencies and their versions, which are then cross-referenced for known vulnerabilities with npm audit. This tool generates extensive reports of vulnerabilities with severity levels as well as suggested updates, which are logged for future analysis and fixes.

2) LAM-Py:

Python projects are identified using two primary methods: detecting virtual environments and searching for `requirements.txt` files in project directories. Virtual environments provide isolated Python environments, preventing global libraries from interfering with the identification of project-specific dependencies. To confirm

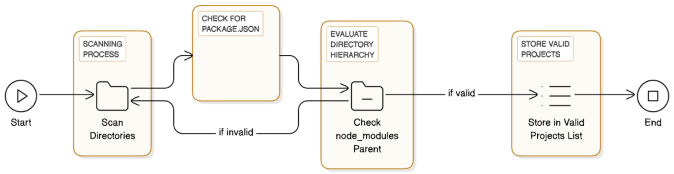

Fig. 1: LAM-Node Scanning Workflow

a directory as a Python project, the detection script checks for the `activate.sh` file, commonly found in Python virtual environments.

If a `requirements.txt` file is missing, the system generates one by activating the virtual environment and running the command `pip freeze > requirements.txt`.

This captures a comprehensive list of installed dependencies and their versions, creating a record necessary for vulnerability analysis. Whether generated or pre-existing, the `requirements.txt` file is parsed to produce an organized list of dependencies.

The system uses Ochrona, a Python-specific vulnerability scanning tool, for vulnerability identification. Ochrona provides detailed information on each detected vulnerability, including CVE details, severity levels, and recommended remediation steps, as emphasized in the work of Benthin Sanguino and Uetz [2]. This allows quick access and tracking of vulnerability data for every Python project.

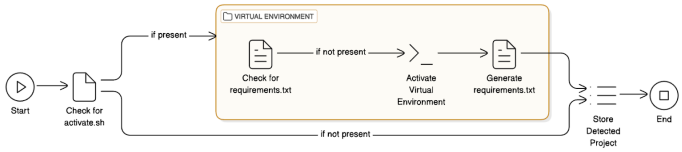

Fig. 2: LAM-Py Scanning Workflow

The system gathers all key vulnerability data after it identifies and examines all the Node.js and Python projects. This includes how severe the issues are, CVE IDs, fixes to consider, and available dependency updates. It then puts this info into a well-structured JSON file. This JSON file gives a full picture of each project's security status, with specific fields for each vulnerability detail. These fields cover the dependency name, version, CVE specifics, how serious the issue is, and what steps to take. By arranging the data in a standard JSON format, the system makes sure data handling is smooth and fits into the main monitoring setup. It also helps with getting reports on each device. This local JSON storage gives admins and developers fast access to useful vulnerability data. This allows them to act and fix issues in their projects.

B. Server-Level Architecture for Distributed Monitoring

Server-level architecture shown in Fig. 3 provides distributed scanning of software vulnerabilities across numerous worker nodes, providing a scalable and centralized mechanism of dependency security management. Comprising a Master System and multiple Worker Nodes, the architecture facilitates continuous scanning of vulnerabilities and data collection from diverse projects in heterogeneous environments. Decentralization of detection responsibilities with uniform surveillance makes the system effective in security analysis without sacrificing flexibility.

1) Master System:

The master system serves as the central hub for coordinating vulnerability data gathering, result storage, and offering a safe interface for viewing reports. It makes the master node to see vulnerability reports and check the security status of all worker nodes connected. The master system comprises these sub-modules.

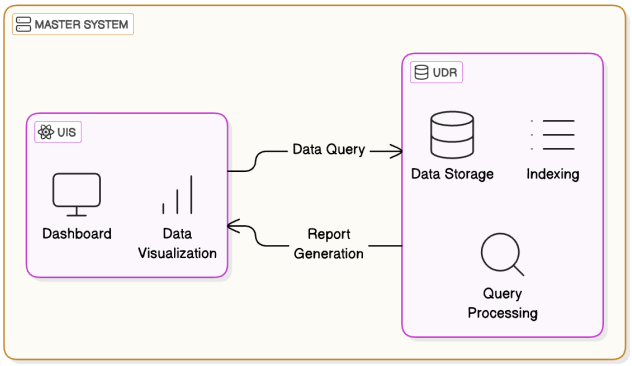

Fig. 3: Master system sub-modules and architecture

User Interface Subsystem (UIS):

The User Interface Subsystem (UIS) is a React web application that is created to offer a user-friendly dashboard for the master node (admin). The subsystem is the major interface for interaction with the centralized monitoring features of the system. It enables the admin to achieve fine-grained information about the security status of all worker nodes and projects that are connected. The UIS provides all the detailed information on vulnerabilities at the level of the individual project including dependency information, identified vulnerabilities along with their severity, CVE IDs, and suggested remedial actions. This makes it possible for the master node to rank and remediate vulnerabilities on the basis of severity, although it does so only at the project level. It also aggregates this data into a single security status overview on the entire ecosystem, from which it takes summary metrics, trends in vulnerability, and history records for compliance purposes into dashboards.

Unified Data Repository (UDR):

The Unified Data Repository (UDR) is a centralized MongoDB database that acts as the primary storage for all system data. It is designed to handle large-scale data storage needs while ensuring fast and reliable access for analysis and reporting. The UDR organizes and indexes vulnerability data, particularly CVE identifiers, along with associated severity ratings, dependency versions, and remediation steps. This indexing ensures that the system retrieves necessary information rapidly whenever an on-demand query or report runs. Such centralized management for maintaining large-scale data about dependencies is cost-effective as highlighted by Latimer et al. [7]. The UDR keeps a record of old vulnerability data and project metadata so that the system can analyze trends across time. This functionality is critical to compliance audits, to conduct post-mortem examinations of incidents, and to prove improvements in the security posture.

2) Worker Nodes

Each node performs local vulnerability scans on projects within its environment, reporting findings back to the master system. A System Orchestration Service (SOS) is deployed on individual workstations

or servers; these systems enable decentralized scanning. The main components of each node include these sub-modules.

Scan Orchestration Service (SOS):

The Scan Orchestration Service (SOS) is a background service that schedules and triggers scans at predefined intervals, ensuring consistent and automated monitoring of dependencies, as shown in Fig. 4. Powered by Cron, SOS initiates scans at regular intervals (e.g., every six hours) without requiring manual intervention. This ensures continuous monitoring and up-to-date vulnerability tracking. SOS is designed to avoid interference with resource-intensive processes, ensuring that scans do not impact system performance during peak workloads. Administrators can configure scan frequencies to suit specific needs, such as prioritizing high-risk environments with more frequent scans or reducing intervals for stable setups.

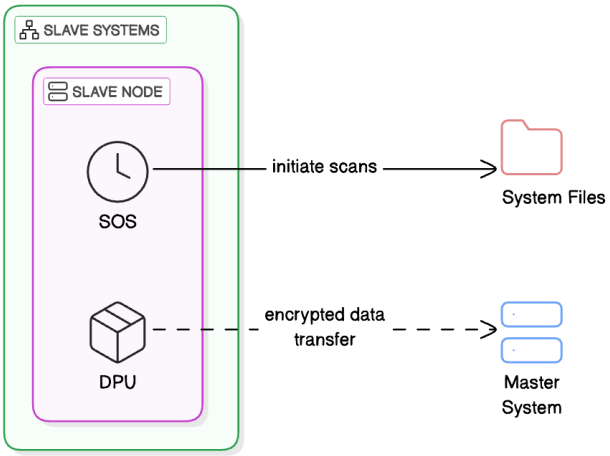

Fig. 4: Worker nodes sub-modules and architecture

Data Packaging Unit (DPU):

The Data Packaging Unit (DPU) is responsible for preparing scan results for secure transmission to the master system. It makes sure that the data is standardized and ready for central processing. It structures those results into well-defined JSON format. This format includes its name, version, CVE Id and severity levels. The DPU also employs encryption protocols like AES to ensure that there is secure communication between salve and master system. Additionally, timestamps and uuids are kept, so that the system keep track of reports effectively even in case of partial data loss or transmission errors.

IV. System-Level Vulnerability Management

In addition to project-level scanning, our system also brings scanning for system-level packages in both macOS and Linux environments. When examining system-level vulnerabilities, we must consider various factors to avoid altering any critical files, especially since we have sudo access. This module addresses vulnerabilities inherent in system packages installed through APT, DNF, Pacman, Homebrew, and MacPorts.

The system uses detection tools like Trivy and CycloneDxGen to scan for installed packages against the latest CVE databases. This integration in our system allows for the detection of direct vulnerabilities as well as transitive dependencies that could potentially introduce security threats [14]. For this to happen properly, we have created optimized scripts that account for numerous edge cases, preventing any impact on critical files. The system creates formatted reports containing severity levels and suggested mitigation steps.

V. Automated Patching Mechanism

The Automated Patching Mechanism is an intelligent module in our system. It is designed to streamline the process of automatically updating vulnerable packages on a project level. But this aspect has many considerations to be done before patching the dependencies. The main problem statement is to make sure that after we have patched a vulnerability, it should work properly and keep the project stable. So, we integrated Docker containers and make use of large language model (LLM) analysis with automated testing to proceed with such a mechanism.

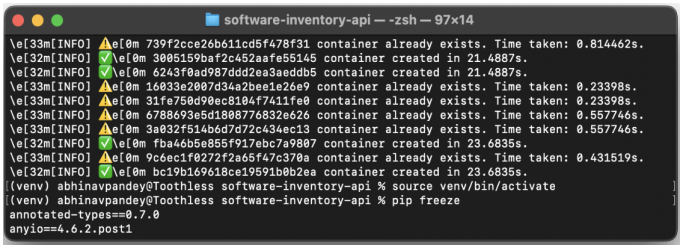

Fig. 5: Docker file generation through LLM

Firstly, we have to analyze whether the project is stable. For that, there needs to be a proper build to do so with minimal manual intervention. We are just considering Node and Python projects for this automation as of now. The system has a segregation tool, which divides the projects based on two criteria. First, if the project already has a build setup in Docker, else it falls in the other category. If there is no build file setup, the system makes use of an in-house deployed small language model (can be configured as per requirements). The inputs to the model can be package.json, readme.md, requirements.txt, etc. The output is a structured Docker file that creates a build for the project, installing all the required dependencies and smartly analyzing the criteria for a build.

To tackle security loopholes effectively, Docker is created such that it mounts the project, installs all the dependencies and adds build scripts along with test cases to ensure stability. The testing is done in an isolated Docker container, where the build is set up. Updates are tested in a sandboxed Docker environment before they are applied to avoid possible conflicts with current configurations. This sandboxed solution prevents updates from introducing instability into the production environment [16]. The result is evaluated through smoke tests and checkpoints, validating a stable build for the project.

LLM Integration for Dockerfile Generation:

- The LLM used is **GPT-4o-mini**, accessed via the OpenAI API.
- Inputs include project files such as `package.json`, `requirements.txt`, and `README.md`.
- The model is prompted to generate a minimal Dockerfile that installs dependencies and builds the project.
- It is used in a stateless, on-demand fashion, with request-response API calls.
- Output is parsed and validated syntactically before containerized execution.
- Model size: 8 billion parameters, Context length: 128,000 tokens
- Temperature: 0.3 (To maintain a deterministic output)

This integration supports dynamic and context-aware Dockerfile creation and reflects modern cloud-native approaches [23], [24].

A. Multi-Layered Validation

Following compatibility checks, the system performs a three-stage validation:

- Unit Tests: Ensures basic functionality is intact.
- Smoke Tests: Detects regressions caused by the update.
- Security Compliance: Assesses risk factors using AI-generated test cases.

This layered approach improves software stability by minimizing the risk of post-update failures.

VI. Results

The efficiency of the presented system was verified through a series of controlled tests that were designed to evaluate different aspects of performance. We tried to simulate a real-world scenario across different hardware configurations. We have defined proper metrics to measure the accuracy of the system using our own dataset, which has many vulnerabilities classified by type and severity. Then, finally, we analyzed the overall system efficiency in comparison with industry standard tools.

A. Dataset and Evaluation Metrics

For controlled testing, we have created a custom dataset that consists of 30 named software project repositories based on Python and JavaScript technologies. These repositories were created to resemble real-world applications.

In these 30 projects, a total of 150 known vulnerabilities were manually injected. These vulnerabilities were taken directly from the National Vulnerability Database, National Institute of Standards and Technology [18]. Each vulnerability instance was inserted into the corresponding dependency files or code components, aligned with known common vulnerabilities and exposures (CVE) patterns.

1) Types of Vulnerability and Severity Levels

The vulnerabilities in the dataset were grouped into 5 major security categories and 3 levels of severity. They are as follows:

By Vulnerability Type:

- Outdated Dependencies: 60 instances
- Remote Code Execution (RCE): 20 instances
- SQL Injection and Data Exposure: 18 instances
- Cross-Site Scripting (XSS): 12 instances
- Authentication and Access Control Flaws: 40 instances

By Severity:

- High Severity: 45 vulnerabilities
- Medium Severity: 65 vulnerabilities
- Low Severity: 40 vulnerabilities

2) Evaluation Metrics

In order to assess the efficiency and effectiveness of the designed system. The following metrics were utilized, which altogether give a full picture of the system's performance in detection precision.

- Detection Rate: The ratio of known vulnerabilities properly detected by the system.
- Average Scan Time (s): The duration for analysis and generation of vulnerability reports per project.

B. Validation Methodology

VULNDET was run to validate the system against this dataset, and the outputs were compared systematically with the ground truth. For benchmarking, two industry tools—Dependabot and Nexus IQ—were used for analysis. Spot-checks were manually performed to ensure no false positives were missed with individual detection results. The results of performance and outputs from detection verify that VULNDET is a reliable and efficient vulnerability detection framework.

C. Benchmark Comparison

The comparison was made between our system, VULNDET, and two popular tools: Dependabot and Nexus IQ. The findings are presented in Table I.

TABLE I: Performance Comparison of Vulnerability Detection Tools

Metric	VULNDET	Dependabot	Nexus IQ
Detection Rate (%)	91.8	88.7	90.4
Average Scan Time (s)	60.4	72.8	58.9

To gain more insight into tool performance by vulnerability type and severity, we categorized detection counts for each of the three tools. Table II and Table III provide these findings.

TABLE II: Detection by Vulnerability Type

Vulnerability Type	Total	Sys 1	Sys 2	Sys 3
Outdated Dependencies	60	59	56	58
Remote Code Execution (RCE)	20	18	17	19
Others (SQLi, XSS, Auth Flaws)	70	61	60	59
Total Detected	150	**138**	**133**	**136**

TABLE III: Detection by Vulnerability Severity

Severity	Total	VULNDET	Dependabot	Nexus IQ
High	45	44	41	43
Medium	65	60	58	59
Low	40	34	34	34
Total Detected	150	**138**	**133**	**136**

This breakdown highlights VULNDET's strong performance across virtually all types of vulnerabilities, especially in its ability to catch outdated dependencies and RCE vulnerabilities. Though all three are good at handling high-severity problems, VULNDET enjoys a steady lead even with medium- and low-severity vulnerabilities, making it justify its worth in a thorough codebase review.

D. Insights and Interpretation

This shows the detection rates and scan times across our dataset fed into those repositories. The 3 systems are here named as: System 1 (VULNDET), System 2 (Dependabot), and System 3 (Nexus IQ). As shown in Fig. 6, System 1 always shows higher detection rates with less variability than the other two systems. But if we observe Fig. 7, it highlights that System 3 has the fastest average scan time, while System 2 exhibits the longest. These plots give us an indication of the trade-offs between detection performance and efficiency for each of these systems.

The results demonstrate that VULNDET achieves a higher detection rate than both competitors, with a detection accuracy of 91.8%. This reflects the system's strength in identifying a wide variety of vulnerabilities, especially those introduced via outdated dependencies and weak access controls.

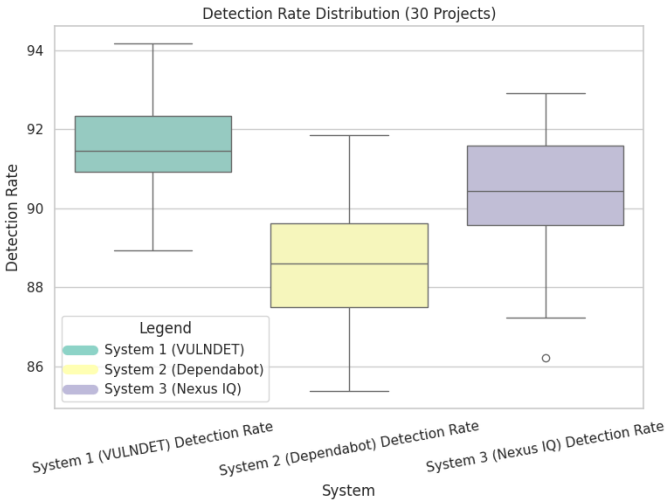

Fig. 6: Detection Rate Distribution

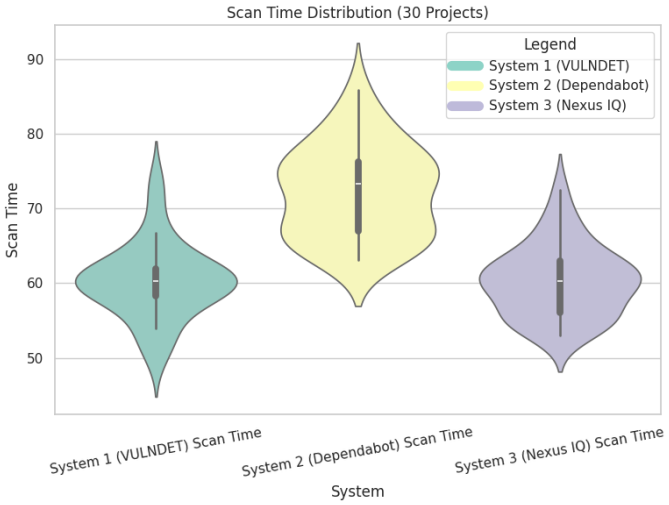

Fig. 7: Scanning Time Distribution

The high detection rate achieved by the system can be credited to a number of important design choices. The first thing to consider is that the system was tested against a well-filtered dataset of project repositories which comprised a realistic and varied set of security vulnerabilities. This ensures that the system was tested against real-world scenarios present in actual codebases. Then, we are utilizing a hybrid detection approach using signature-based matching and optimized scripts that can detect dependencies on both the code and system levels. Although the scan time is a bit greater than Nexus IQ, it is still within reasonable parameters and significantly faster than Dependabot. The compromise between detection depth and performance makes it ideal for both development and production environments where timely feedback is necessary.

Although VULNDET exhibits a slightly longer scan time than Nexus IQ (60.4s vs. 58.9s), this trade-off is a result of its more thorough and distributed scanning architecture. VULNDET performs deeper scanning through language-specific modules (LAMs) like LAM-Py and LAM-Node, which incorporate tools such as Ochrona and npm audit. These scans include transitive dependency analysis, which can increase scan time but also boosts detection accuracy—particularly for medium- and low-severity vulnerabilities II. Unlike Nexus IQ, VULNDET employs JSON-based detailed reporting per project, which slightly increases processing time but offers richer metadata for remediation. But this marginal time cost is also dependent on the coverage and environments on which the tests were taken.

E. Scope for Research and Security Implications

During the evaluation phase, the implementation of the automated patching mechanism showed promising capabilities with the approach. Streamlining such a remediation process through isolated container environment testing is stable. But the system is dependent on LLMs for the patching logic. The whole logic's behavior displays clear potential for expansion into more intelligent and autonomous patch generation systems.

If such mechanisms are perfected and supported by strong validation protocols, they could significantly reduce the time-to-fix for high-severity vulnerabilities, particularly in large-scale or poorly maintained projects. This opens a broad scope for research in areas such as machine learning-based code repair, semantic patch generation, and trust-aware patch deployment systems.

There are a few things that should be considered when dealing with such patching mechanisms. Some patches, while technically correct, caused unstable builds or compatibility problems through breaking changes or the absence of downstream support. So the concern is that without proper testing, validation and rollback mechanisms, automated patching can not be stable. If addressed properly, the integrity and security of the system are safeguarded.

VII. Limitations and Future Directions

The suggested vulnerability detection system currently suffers from several issues. It offers a strong dependency security management framework, and yet, the system's overall vulnerability detection capability is scaled and not very efficient presently. At the moment, the system is focused on the vulnerability detection of those projects that use Python, JavaScript, Go, or Java. As mentioned, these technologies are currently supported, but the support is extensible for further languages and frameworks.

The centralized architecture of the system introduces a potential single point of failure: the master node aggregates data from multiple worker nodes. If the master node or its MongoDB instance is down, the entire system cannot monitor and report on vulnerabilities, thereby compromising the reliability of this system, especially in environments with high security standards. Additionally, while the system has a basic notification capability, it does not offer user-specific, detailed messages that might be customised according to the importance of vulnerabilities or project roles. The absence of personalised alerts could cause remediation efforts to be delayed in situations where high-priority vulnerabilities need to be addressed quickly.

As the quantity of data increases with that of monitored projects, a MongoDB database may face a performance bottleneck in retrieving information, thus slowing down responses in a large-scale distributed environment. In addition to this, the system under consideration is only available to macOS and Linux environments, therefore excluding Windows-based projects. Therefore, as suggested by Johnson and Clarke [11], expanding this system to serve Windows-based environments would be one of the top objectives. Because of this platform dependency, the system is less flexible and less suitable for businesses that need cross-platform support.

Future development is planned in a number of areas to address existing limits and enhance the system's capabilities. More languages and package managers, including Ruby, PHP, and others, would increase the system's suitability for a wider variety of applications. Integrating tools like Bundler-audit or Snyk could provide

multi-language support, increasing the system's versatility in mixed environments. Additionally, implementing redundancy and failover mechanisms, such as a clustered setup or a backup master node, would improve reliability by mitigating the single point of failure.

Another key improvement involves enhancing the notification system. Introducing user-specific, role-based notifications and integrating with communication platforms like Slack, email, or SMS would allow administrators to receive targeted alerts. This will increase the response rate, especially in high-security environments whereby the need for quick identification of vulnerabilities is a priority. Extending the compatibility of the system to add a Windows environment will increase flexibility, where the organization can have cross-platform infrastructural organizations fully utilize the system.

VIII. Conclusion

This paper introduces a vulnerability detection system that offers an effective way to manage dependency security in Java, Python, JavaScript and Go environments. The solution provides a centralized, scalable way of tracking vulnerabilities across multiple projects and devices by combining generated or capabilities with a distributed, master-slave design. With automated dependency checks, secure data storage, and continuous monitoring, this system empowers administrators to proactively address security risks in software dependencies, supporting overall organizational security.

Despite its current shortcomings, such as platform limitations and dependence on certain vulnerability scanners, the system provides a valuable basis for scale vulnerability identification. Its design may be enhanced with more languages, improved notification functions, and advanced machine-learning capabilities. Future research will focus on enhancing resilience, expanding platform compatibility, and integrating real-time monitoring to meet the demands of dynamic, high-security situations.

This technique is a majorly the biggest advance in vulnerability management of open-source dependencies in order to meet the needs of current software development, which incorporates important factors such as security, scalability, and flexibility. The fundamental framework provided here sets it for future advances in automated and distributed vulnerability management across many diverse computing platforms since the landscape of software security will continue to change.

References

[1] J. Shin, Z. Wang, Z. Chen, and Y. Cheng, "VulPecker: An automated vulnerability detection system based on code similarity analysis," Proceedings of the 31st Annual Computer Security Applications Conference, 2015, pp. 146–155.

[2] L. A. Benthin Sanguino and R. Uetz, "Software Vulnerability Analysis Using CPE and CVE," arXiv preprint arXiv:1705.05347, 2017.

[3] F. Brown, P. Black, and L. Choo, "Code-based vulnerability detection in Node.js applications," Proceedings of the 27th International Symposium on Software Testing and Analysis, 2018, pp. 114–122.

[4] M. Alfadel, D. E. Costa, and E. Shihab, "On the discoverability of npm vulnerabilities in Node.js projects," ACM Transactions on Software Engineering and Methodology, vol. 32, no. 4, 2023, pp. 1–27.

[5] GitHub, "GitHub Dependabot Documentation," Accessed on: [insert date], Available: https://docs.github.com/en/code-security/supply-chain-security/keeping-your-dependencies-updated-automatically.

[6] "Ochrona: A Python package vulnerability detection tool," Accessed on: [insert date], Available: https://pypi.org/project/ochrona/.

[7] K. Latimer, M. J. Harrold, and D. Rosenblum, "Automated extraction and analysis of vulnerability patterns in software dependencies," IEEE Transactions on Software Engineering, vol. 47, no. 3, pp. 512–529, March 2021.

[8] N. Semenov, I. Ivanov, and D. Klebanov, "A distributed architecture for real-time vulnerability scanning in cloud-native environments," Proceedings of the IEEE International Conference on Cloud Computing Technology and Science, 2020, pp. 71–78.

[9] S. Miramirkhani, M. Protsenko, and A. Razavizadeh, "A hybrid approach to vulnerability management in open-source ecosystems," Journal of Systems and Software, vol. 169, pp. 110711, 2021.

[10] J. Derr, E. Manes, and A. Parnin, "Vulnerabilities in open-source software: Can vulnerability prediction models help?" Proceedings of the ACM Conference on Computer and Communications Security, 2019, pp. 1189–1201.

[11] T. Johnson and R. Clarke, "Automated dependency management in JavaScript and Python: Challenges and advances," ACM Computing Surveys, vol. 53, no. 2, pp. 29:1–29:35, Apr. 2020.

[12] Z. Peng, X. Li, and Y. Xu, "Real-time monitoring and vulnerability detection in microservices with distributed tracing," IEEE Access, vol. 9, pp. 154867–154879, 2021.

[13] C. Cesarano, V. Andersson, R. Natella, and M. Monperrus, "GoSurf: Identifying Software Supply Chain Attack Vectors in Go," Proceedings of ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses, 2024. Available: https://doi.org/10.48550/arXiv.2407.04442.

[14] Semenov, N., Ivanov, I., Klebanov, D., "A distributed architecture for real-time vulnerability scanning in cloud-native environments," IEEE Cloud Computing, 2020.

[15] Miramirkhani, S., Protsenko, M., Razavizadeh, A., "A hybrid approach to vulnerability management in open-source ecosystems," Journal of Systems and Software, vol. 169, 2021.

[16] Derr, J., Manes, E., Parnin, A., "Vulnerabilities in open-source software: Can vulnerability prediction models help?" ACM CCS, 2019.

[17] Peng, Z., Li, X., Xu, Y., "Real-time monitoring and vulnerability detection in microservices with distributed tracing," IEEE Access, 2021.

[18] Harold Booth (2015), National Vulnerability Database, National Institute of Standards and Technology, https://nvd.nist.gov/ (Accessed 2025-04-13)

[19] K. Achuthan, S. SudhaRavi, R. Kumar, and R. Raman, "Security vulnerabilities in open source projects: An India perspective," in *Proc. Int. Conf. on Information and Communication Technology (ICoICT)*, 2014, pp. 255–260, doi: 10.1109/ICoICT.2014.6914033.

[20] M. R. Parimi and S. Babu, "Critical analysis of software vulnerabilities through data analytics," Proc. Int. Conf. Ind. Eng. Oper. Manage., Dubai, UAE, Mar. 10–12, 2020, pp.

[21] A. M, S. R and S. G, "Optimizing Smart Contract Security: A Cost-Sensitive Graph Neural Network Approach for Vulnerability Detection," 2024 International Conference on Electronic Systems and Intelligent Computing (ICESIC), Chennai, India, 2024, pp. 191-195, doi: 10.1109/ICESIC61777.2024.10846442.

[22] S. Chandran, Hrudya P, and P. Poornachandran, "An efficient classification model for detecting advanced persistent threat," *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Kochi, India, 2015, pp. 2001–2009, doi: 10.1109/ICACCI.2015.7275911.

[23] R. V. Savant, S. N. Sunder, S. Seshadri, N. Panda and S. M. Rajagopal, "Cloud-Native CDN Monitoring Using CI/CD," 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT), Kamand, India, 2024, pp. 1-9, doi: 10.1109/ICCCNT61001.2024.10724159.

[24] M. Pandey, N. S. S. Kunda, P. KC and K. D. Kumar, "Creating a Virtual Literary Hub: Leveraging AWS for Deploying an Application in the Cloud," 2025 3rd International Conference on Smart Systems for applications in Electrical Sciences (ICSSES), Tumakuru, India, 2025, pp. 1-6, doi: 10.1109/ICSSES64899.2025.11009959.