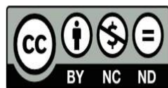
Indian Journal of Computer Science and Technology
https://www.doi.org/10.59256/indjcst.20260502084
Volume 5, Issue2 (May-August 2026), PP: 741-752.
www.indjcst.com

ISSN No: 2583-5300

AI-Powered Vulnerability Scanner for Spring Boot Applications

Saad Mirza[1], Ashwini Kadam[2], Shreya Sase[3], Apoorva Kulkarni[4], Dr. S. K. Wagh[5]
[1,2,3,4] Department of Computer Engineering, M. E. S. Wadia College of Engineering, Savitribai Phule Pune University, Pune, Maharashtra, India.
[5]Professor, Department of Computer Engineering, M. E. S. Wadia College of Engineering, Savitribai Phule Pune University, Pune, Maharashtra, India.

To Cite this Article: *Saad Mirza[1], Ashwini Kadam[2], Shreya Sase[3], Apoorva Kulkarni[4], Dr. S. K. Wagh[5], "AI-Powered Vulnerability Scanner for Spring Boot Applications", Indian Journal of Computer Science and Technology, Volume 05, Issue 02 (May-August 2026), PP: 741-752.*



Copyright: ©2026 This is an open access journal, and articles are distributed under the terms of the Creative Commons Attribution License; Which Permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Abstract: *Modern web-based applications built with Spring Bootframework are widely used, but they are still vulnerable to common configuration and coding mistakes. We present an AI-assisted scanner that combines static rule-based analysis with lightweight dynamic API probes and attaches concise AI explanations for each finding. The system runs locally by default using a WebLLM (WebGPU-enabled) model for privacy; a cloud model may be used optionally for short, redacted snippets when higher-quality reasoning is required. We describe detector design, runtime probes, the explanation pipeline, and a prototype implementation. An initial evaluation on small Spring Boot samples demonstrates the workflow and trade-offs. The focus is practical developer help and privacy-aware explainability rather than claiming enterprise-grade coverage.*

Keywords: *Spring Boot security, static analysis, dynamic analysis, explainable AI, WebLLM, WebGPU.*

I. INTRODUCTION

Web applications are being used in almost every field which has made security an important aspect of software development. Spring Boot is one of the most popular frameworks for developing Java-based web applications because it allows developers to build applications quickly and efficiently and it is easier for beginners. However, many applications suffer from security issues such as exposed endpoints, insecure con-figurations, weak authentication mechanisms, and hardcoded credentials. If these vulnerabilities are not detected early, they can lead to serious security breaches.

The idea behind this project is to develop an AI-Powered Vulnerability Scanner that can help developers identify secu-rity weaknesses in Spring Boot applications. The proposed system combines static analysis and dynamic analysis to detect vulnerabilities from both the source code and the running application. In addition to finding vulnerabilities, the system uses Artificial Intelligence to explain the detected issues in simple language and provide suitable recommendations for fixing them.

Unlike many existing tools that generate complex technical reports, the proposed solution focuses on making security findings easier to understand. The project also aims to protect user privacy by supporting local analysis, ensuring that sensitive source code does not need to be uploaded to external servers. The overall goal is to create a practical, intelligent, and user-friendly security tool for developers and students.

When we were doing our internship , we were using Spring Boot. As we were learning, we had to continuously ask our seniors whenever facing any security issues. And whenever we used online scanners, we could not understand the findings. The realization that this problem is faced by so many students and new developers; it became our primary motive to find solution which lead to the core idea of this project.

To overcome these challenges, we came up with AI-powered vulnerability scanner which enables developers to perform local security scans with out compromising code privacy. By combining AI-report generation and automated fix sugges-tions, the tool ensures secure, compliant, and efficient solution. As technology evolves everyday, cyber security threats are increasing as well. Easiest target for attackers are web applica-tions. Therefore it is essential for organizations to secure their web applications. Many students, professors and researchers have carried out studies focusing on these problem, improv ing vulnerability detection, secure coding practices and auto-mated security assessment techniques. These all contributed in developing the proposed solution AI-Powered Vulnerability Scanner for Spring Boot Applications..

The main contributions of this work are as follows:

- A hybrid security scanning approach for Spring Boot applications combining static analysis and lightweight dynamic probes.
- An AI-based explanation module that converts technical findings into simple and understandable descriptions.
- A privacy-focused design that supports local execution with an optional cloud-based fallback.

II.RELATED WORK

The evolution of automated security auditing has transi-tioned from simple pattern matching to complex heuristic engines. However, the specific intersection of Spring Boot's "convention-over-configuration" model and local AI-driven explainability remains an underexplored domain. We catego-rize the existing literature into three primary pillars: traditional analysis frameworks, AI-augmented security, and privacy-preserving local inference.

A. Traditional SAST and DAST for Java Ecosystems

Static Application Security Testing (SAST) has long relied on Abstract Syntax Tree (AST) analysis and Data Flow Equations. Tools like FindSecBugs and SonarQube utilize pre-defined bug patterns to identify vulnerabilities in Java bytecode and source. While effective for spotting standard injection

flaws, these tools often struggle with the "Implicit Configu-ration" nature of Spring Boot. For instance, a vulnerability might only exist if a specific @ConditionalOnProperty annotation is absent—a context that many general-purpose SAST tools fail to parse accurately.

Dynamic Application Security Testing (DAST) tools, such as OWASP ZAP and Burp Suite, address this by probing the application at runtime. However, in a microservices context, DAST tools require a fully orchestrated environment, which is often too heavy for a developer's local machine during early-stage coding. Our work seeks to lower this barrier by performing lightweight, targeted probes specifically for Spring-managed endpoints like Actuators.

B. Large Language Models in Vulnerability Remediation

The integration of Large Language Models (LLMs) into the software development life cycle (SDLC) has shifted the focus from simple detection to automated remediation. Research into "Explainable AI" (XAI) for security suggests that developers are more likely to patch a vulnerability if they understand the underlying exploit mechanism. Recent studies have demon-strated that models like GPT-4 can suggest code-level fixes for Common Weakness Enumerations (CWEs) with high accuracy. However, a major limitation identified in the literature is the "Context Window" problem—providing the LLM with enough code to be accurate without overwhelming the model's memory. Our approach addresses this by using the static analyzer to "crop" relevant code snippets, ensuring the LLM receives only the necessary functional context for a high-quality explanation.

C. Privacy-Preserving and Local Security Tooling

Data residency and the protection of intellectual property (IP) are significant hurdles for cloud-based security tools. In enterprise and academic settings, uploading proprietary source code to a 3rd-party LLM provider is often prohibited. This has spurred interest in local-first AI architectures.

Projects like PrivateGPT and Local-LLM frameworks have proven that quantized models can perform reasoning tasks on consumer-grade hardware. By leveraging WebLLM and WebGPU acceleration, our system bypasses the need for a centralized server entirely. We build upon the work of Trans-formers.js to bring the "Security Engineer in a Box" concept directly to the browser, ensuring that the vulnerability analysis loop remains strictly within the developer's local perimeter.

D. Summary of the Research Gap

While existing literature covers the "How" of detection and the "What" of AI reasoning, there is a clear gap in providing a **Spring-specific hybrid scanner** that operates with **zero-data-leakage**. Most current tools either prioritize enterprise-scale coverage at the cost of privacy or provide local scanning without an accessible explanation layer. Our system bridges this gap by marrying rule-based precision with local AI-driven intuition.

III. PROPOSED SYSTEM AND METHODOLOGY

Figure 1 shows the overall architecture.

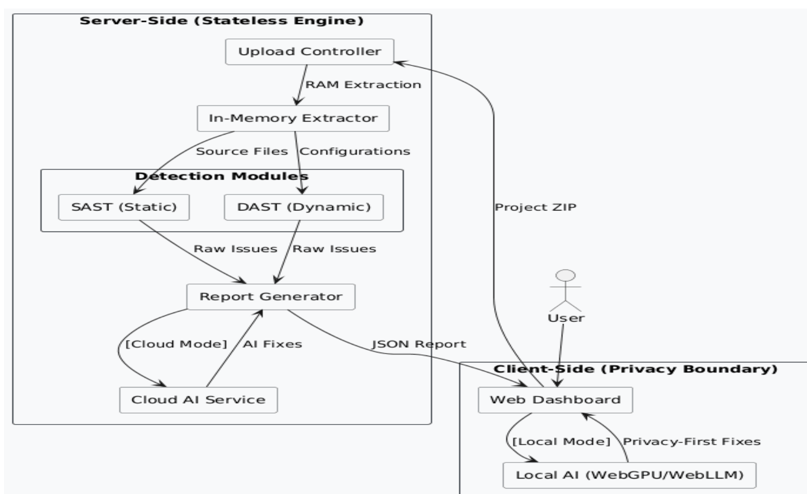


Fig. 1: Overall system architecture of the AI-assisted Spring Boot security scanner

A. Architecture Design Details

The system architecture follows a modular design where each component is responsible for a specific task. This separation of concerns improves maintainability and allows independent development of modules.

The frontend provides a simple interface for uploading projects and viewing reports. It communicates with the back-end, which orchestrates the entire scanning process.

The ingest module handles file extraction and preprocessing. It ensures that only relevant files are passed to the analysis components. The static analyzer processes source code and configuration files, while the dynamic analyzer interacts with application endpoints.

The AI module acts as an interpretation layer, transforming technical findings into human-readable explanations. This improves usability and reduces the effort required to understand vulnerabilities.

The reporting module consolidates all results and generates structured outputs in both JSON and PDF formats. The system avoids long-term storage of user data, ensuring privacy and security.

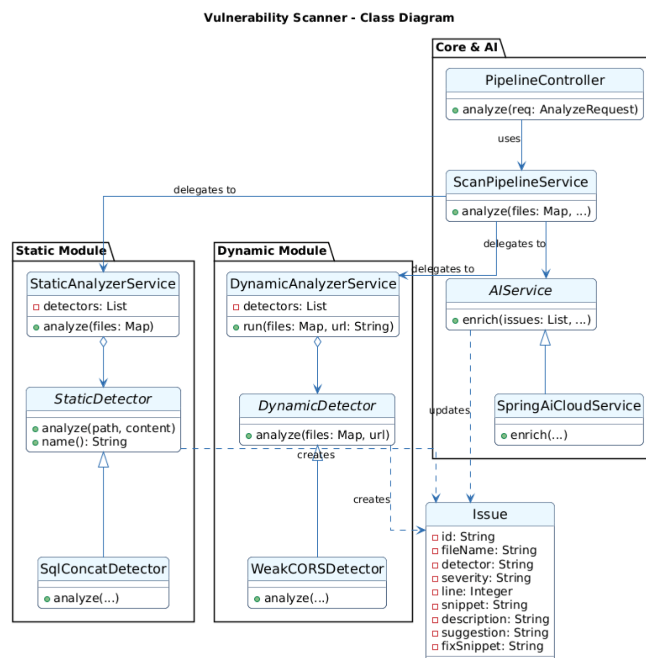


Fig. 2: Class diagram of the proposed AI-assisted vulnerability scanner.

B. System Workflow

The working of the proposed system follows a structured pipeline, where each stage processes the output of the previous stage. The overall flow is designed to keep the system simple, modular, and easy to understand, while still covering both static and runtime analysis.

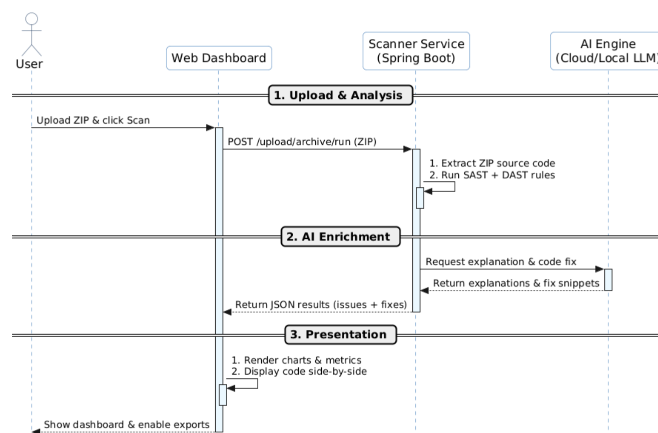


Fig. 3: Interaction diagram of the proposed AI-assisted vul-nerability scanner.

1) **Project Upload:** The process begins when the user uploads a Spring Boot project through the web dashboard. The system supports uploading a compressed project archive (ZIP) or a repository link. This step acts as the entry point to the scanning pipeline and ensures that the input is standardized before further processing.
2) **Upload Service and Extraction:** Once the project is received, the upload service stores it temporarily and passes it to the extractor module. The extractor unzips the project and identifies relevant files such as Java source files, configuration files

(.properties, .yml), and resource files. Only neces-sary files are processed further to avoid unnecessary overhead.

3) **Static Analysis Phase:** In this stage, the static analyzer scans the extracted files without executing the application. A set of predefined detectors is applied to identify com-mon security issues. These include problems such as SQL query concatenation, hardcoded secrets, weak hashing usage, insecure JWT handling, and cross-site scripting risks. Each detector works independently and records findings along with file location and context.

   This phase is fast and helps identify a large number of potential issues early, but it may also include false positives since the code is not executed.

4) **Dynamic Analysis Phase:** After static analysis, the system performs lightweight runtime checks. These tests in-teract with the running application or a sandboxed instance using HTTP requests. The goal is to verify whether certain vulnerabilities are actually exploitable.

   Examples include checking for unprotected admin end-points, weak CORS configurations, sensitive data exposure in API responses, and basic fuzzing of request parameters. This step helps confirm real vulnerabilities and reduces false positives generated during static analysis.

5) **Data Aggregation:** Results from both static and dynamic analysis are collected and merged into a unified structure. Each finding is assigned basic metadata such as type, location, and severity estimate. This combined dataset is then passed to the AI module for further processing.

6) **AI-Based Explanation:** The AI engine processes each finding and generates a human-readable explanation. Instead of just listing the issue, it explains why the vulnerability is risky and how it can affect the system. It also suggests simple remediation steps that developers can follow.

   By default, this step runs locally using a WebLLM-based model, ensuring that source code does not leave the system. If required, the user can enable a cloud-based model, where only small and redacted snippets are sent to improve explanation quality.

7) **Report Generation:** Once all findings are processed, the report generator creates structured outputs. The results are saved in formats such as JSON for machine processing and PDF for human readability. Each report includes vulnerability details, severity, explanation, and suggested fixes.

8) **Result Visualization:** Finally, the generated report is dis-played on the web dashboard. Users can review vulnerabilities, understand the issues, and download the report for further use. This completes the full scanning cycle.

C. Overall, this workflow ensures that the system remains lightweight while still providing meaningful security insights. The combination of static detection, runtime validation, and AI-based explanation makes the process both practical and easy to use Detailed Workflow Explanation

The workflow of the system is designed as a sequential pipeline where each stage transforms the input and passes structured output to the next stage. The system follows a step-by-step pipeline: each stage takes the input, turns it into a clear structured result, and then hands it off to the next stage. It makes the system easier to expand and debug by keeping pieces modular. It starts when someone uploads a Spring Boot project as a zipped file. The system pulls out everything, then keeps only the useful parts, like Java source code and config files such as .properties and .yml. This step filters out everything except the files that need checking. During static analysis, every file is checked on its own using a fixed set of detectors.

They spot weak spots by comparing patterns and running structural checks. Because this phase doesn't run the code, it's fast and helps you spot issues early while developing. Dynamic analysis works alongside static analysis by testing how the app behaves while it runs. Send HTTP requests to different endpoints to watch how things behave live. This step checks if the found vulnerabilities can truly be exploited. Once both analysis steps finish, we collect the findings and send them to the AI explanation module. It breaks each finding into easy explanations and suggests fixes, helping developers understand faster. At last, the user gets a clear, organized report.

D. Static Detector Engine

The static detector engine analyzes the source code and con-figuration files without executing the application. It focuses on identifying common security issues using rule-based patterns and simple code checks. Each detector works independently and looks for specific types of vulnerabilities.

The system currently includes the following detectors:

- SQL Concatenation Detector: Identifies SQL queries built using string concatenation, which may lead to SQL injection vulnerabilities.
- Hardcoded Secret Detector: Detects sensitive informa-tion such as API keys, passwords, and tokens directly written in source code or configuration files.
- JWT Secret Detector: Flags weak or hardcoded JWT signing keys that can compromise authentication mecha-nisms.
- Weak Hashing Detector: Identifies insecure hashing algorithms such as MD5 and SHA-1.
- Cross-Site Scripting (XSS) Detector: Detects cases where unvalidated user input may be injected into web responses.
- Command Injection Detector: Identifies situations where user input is directly used in system-level com-mand execution.

- Insecure Deserialization Detector: Flags unsafe deseri-alization practices that may allow execution of malicious payloads XXE Detector: Detects XML parsers that do not dis-able external entity processing, which may lead to data leakage.
- Open Redirect Detector: Identifies redirection logic that accepts user-controlled URLs without validation.
- Insecure Randomness Detector: Detects usage of predictable random generators such as java.util.Random in security-sensitive contexts.

These detectors help in identifying vulnerabilities early in the development stage. However, since the analysis is static and does not execute the code, some findings may not always be exploitable in practice.

E. Detailed Static Analysis Discussion

The static analysis phase plays a critical role in identifying vulnerabilities early in the development lifecycle. Since it does not require execution of the application, it provides fast and efficient feedback to developers.

Each detector is designed to focus on a specific category of vulnerability. For example, injection-related detectors analyze how user input is handled within the code, while configuration-based detectors examine security settings in application files. The use of pattern matching allows the system to detect commonly occurring vulnerabilities with minimal computa-tional overhead. However, static analysis may lack context, which can lead to false positives in certain scenarios. Despite this limitation, static analysis remains highly valuable as it enables early detection of issues and reduces the cost of fixing vulnerabilities later in the development process.

F. Detector Design Strategy

The detector engine follows a modular rule-based design where each detector is responsible for identifying a specific type of vulnerability. The detector engine uses a modular, rule-based approach: each detector is built to find one specific kind of vulnerability. It makes things more flexible, so you can add new detectors without breaking what already works. Each detector works on its own, checking files by searching for patterns and doing quick structure checks. Regular expressions help spot frequent security issues, like hardcoded secrets, un-safe SQL concatenation, and weak or insecure configurations. It keeps things simple and easy to understand. Rather than using complicated analysis, the system uses straightforward detection methods that are easy to grasp and build on later. It works well for classroom learning and for real-life use. Even if rule-based detection misses harder vulnerabilities hidden in deep data paths, it quickly finds the usual problems.

G. Dynamic Analyzer

The dynamic analyzer complements static analysis by test-ing the application at runtime. Instead of scanning code, it interacts with application endpoints and observes actual system behavior.

The following runtime checks are implemented:

Sensitive Data Exposure Detection: Checks API re-sponses for unintended exposure of sensitive information.

- HTTP Fuzzing: Sends modified or unexpected inputs to endpoints to identify missing validation or unstable behavior.
- Weak CORS Detection: Verifies whether the application allows overly permissive cross-origin requests.
- Unprotected Admin Endpoint Detection: Attempts to access sensitive endpoints such as /actuator without proper authentication.

Dynamic analysis helps confirm whether vulnerabilities detected during static analysis are actually exploitable. It also helps identify runtime issues that cannot be detected through code inspection alone.

H. Detailed Dynamic Analysis Discussion

The dynamic analysis component complements static anal-ysis by evaluating the application during execution. This allows the system to observe real behavior and verify whether detected vulnerabilities are exploitable.

The system performs lightweight runtime checks using HTTP-based interactions with application endpoints. These checks are designed to simulate common attack scenarios without causing harm to the application.

Dynamic analysis is particularly useful for identifying issues such as improper access control, insecure configurations, and exposure of sensitive data. By validating findings from static analysis, it helps reduce false positives and improves overall accuracy.

However, the depth of dynamic analysis is intentionally limited to maintain system performance and safety. More advanced runtime testing techniques can be incorporated in future improvements.

I. AI Explainer Module (Detailed)

The AI explainer module is designed to bridge the gap between vulnerability detection and developer understanding. Traditional security tools often provide technical outputs that are difficult to interpret, especially for beginners.

The module processes each detected issue and generates a simplified explanation that describes the nature of the vul-nerability, its potential impact, and recommended remediation steps.

A key design aspect is the local-first execution model. The system uses WebLLM with WebGPU support to perform inference directly in the browser. This ensures that source code remains within the user's environment, preserving privacy.

In scenarios where higher-quality explanations are required, an optional cloud-based model can be used. Before sending data externally, the system redacts sensitive information and limits the size of the input.

The combination of local and cloud-based inference pro-vides a balance between privacy and explanation quality. This approach makes the system both practical and secure for real-world usage.

J. Report Generation and Output

The reporting module is responsible for presenting the results in a structured and user-friendly format. Each detected

vulnerability is recorded along with relevant metadata such as file name, line number, severity level, and confidence score.

The report also includes AI-generated explanations and sug-gested fixes, making it easier for developers to understand and resolve issues. This significantly reduces the effort required to interpret raw vulnerability data.

Reports are generated in both JSON and PDF formats. The JSON format is useful for integration with other tools, while the PDF format provides a human-readable summary that can be used for documentation or review purposes.

The system ensures that reports do not include sensitive information beyond what is necessary. This aligns with the privacy-first design approach and ensures safe handling of user data.

Algorithm 1 Hybrid Scan and Privacy-First Remediation Pipeline

Require: Uploaded project ZIP, User-selected AI Mode (Cloud/Local)
Extract files into memory (Zero-Storage architecture)
Initialize empty *Issues* list
for each *file* in extracted files **do**
Execute Static Analyzers (Regex/Heuristic)
Append findings to *Issues*
end for
Execute Dynamic Config Probes on *file* metadata Append configuration findings to *Issues*
if Mode == Cloud **then** Batch *Issues* into chunks
Query Cloud LLM (via Spring AI) for remediation Update *Issues* with generated fixes
Return enriched JSON report to Client **else if** Mode == Local (Privacy-First) **then**
Return raw JSON *Issues* to Client
Initialize WebGPU LLM Engine in Browser Cache **for** each *issue* in *Issues* **do**
Prompt Local In-Browser LLM with code snippet Render AI-generated fix asynchronously in UI
end for
end if

K. Threat Model

The proposed system focuses on identifying common vul-nerabilities in Spring Boot applications that arise due to insecure coding practices and misconfigurations.

The primary threat actors considered include external at-tackers attempting to exploit exposed endpoints, injection vulnerabilities, and weak authentication mechanisms. These attackers may leverage vulnerabilities such as SQL injection, command execution, insecure deserialization, and cross-site scripting to gain unauthorized access or manipulate application behavior.

The system assumes that the application under analysis is accessible in a controlled environment for dynamic testing. It does not simulate advanced adversaries using multi-stage attacks or zero-day exploits but focuses on practical and commonly observed vulnerabilities.

The threat model also considers risks related to data expo-sure, including leakage of sensitive information through API responses or insecure configuration settings. By addressing these threats, the system aims to improve the baseline security posture of Spring Boot applications.

This simplified threat model aligns with the goal of building a lightweight and practical vulnerability scanner suitable for educational and small-scale development use.

IV. IMPLEMETATION DETAILS

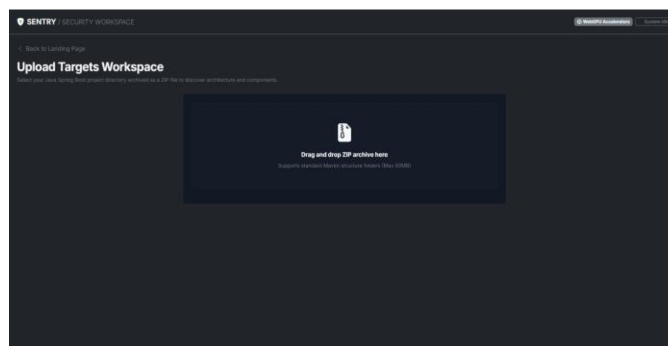


Fig. 4: Upload Screen of AI-assisted vulnerability scanner

.

The system was developed as a lightweight prototype with a focus on practical usability instead of large-scale deployment. The aim was to keep the design simple and modular, so that new detectors or features can be added without affecting existing components. At the same time, the implementation still covers essential security checks relevant to Spring Boot applications.

The backend is implemented using Java 17 and Spring Boot, which handle both orchestration and a basic web interface. For static analysis, the system relies on pattern matching using regular expressions along with simple AST-based checks. This

approach avoids heavy processing while still being effective in identifying common vulnerability patterns.

Dynamic testing is carried out using a configurable HTTP client that interacts with application endpoints. In certain scenarios, sandboxed execution using Docker is used to safely test behaviors that could be risky. This allows runtime validation without impacting the host environment.

The AI explanation module follows a local-first approach. WebLLM and Transformers.js are used with WebGPU support, allowing inference to run directly in the browser. This ensures that source code remains on the local system. When required, a cloud-based model can be used, but only after sensitive information is removed from the input.

The reporting module generates structured outputs in JSON format and also produces human-readable PDF reports using Apache PDFBox or LaTeX-based export.

A. Sample Heuristics

To illustrate how vulnerabilities are detected, a few repre-sentative rules are shown below. In the actual system, these rules are implemented using pattern matching combined with lightweight code structure checks.

1) Hard-coded secrets detector

```
(?i)(api[_-]?key|secret|token|password)\s*[:=]\s*['"]
[A-Za-z0-9\-_]{8,}['"]
```

```
Matcher keyMatch = KEY_ID.matcher(line);
Matcher stringMatch = STRING_LITERAL.matcher(line);

if (keyMatch.find() && stringMatch.find()) {
   issues.add(new Issue(
      "Hardcoded Secret",
      "Sensitive value found in source code"
   ));
}
```

Listing 1: Hardcoded Secret Detection Logic

This detector searches for sensitive identifiers such as API keys, tokens, or passwords and checks whether they are directly assigned string values. Hardcoding such secrets in source code is a common security risk.

This rule identifies variables or configuration entries that appear to contain credentials directly in the code. Instead of deep parsing, it relies on commonly used naming patterns and assignment structures.

2) SQL Injection Detection

```
"SELECT" | "INSERT" | "UPDATE" | "DELETE"
```

```
if (SQL_KEY.matcher(line).find() && line.contains("+")) {
   issues.add(new Issue(
      "SQL Injection Risk",
      "Query built using string concatenation"
   ));
}

if (EXEC_CALL.matcher(line).find() && line.contains("+"))
      {
   issues.add(new Issue(
      "SQL Execution Risk",
      "Execution with concatenated query"
   ));
}
```

Listing 2: SQL Concatenation Detection Logic

This detector identifies SQL queries constructed using string concatenation. Such patterns are commonly associated with SQL injection vulnerabilities, especially when user input is directly appended to query strings.

3) Open CORS configuration detector

```
registry.addMapping("/**").allowedOrigins("*")
```

```
Matcher m = CMD_EXEC_PATTERN.matcher(content);

while (m.find()) {
   issues.add(new Issue(
      "Command Injection Risk",
      "System command execution detected"
   ));
}
```

Listing 3: Command Execution Detection

This detector identifies usage of system-level command execution APIs such as Runtime.exec or ProcessBuilder. If user input is passed into these commands without validation, it can lead to command injection vulnerabilities.

This rule captures overly permissive CORS configurations where requests from any origin are allowed, which can expose the application to cross-origin attacks.

4) Admin Endpoint Detector

```
Matcher m = ADMIN_PATH_PATTERN.matcher(content);

if (m.find() && !AUTH_HINT_PATTERN.matcher(content).find
      ()) {
   issues.add(new Issue(
      "Unprotected Admin Endpoint",
      "No authentication detected"
   ));
}
```

Listing 4: Unprotected Admin Endpoint Detection

This runtime check identifies sensitive endpoints such as admin or management routes and verifies whether proper authentication mechanisms are present. Missing protection on such endpoints can lead to unauthorized access.

B. Privacy and Data Handling

Privacy is treated as a core requirement in the system design. By default, all analysis is performed locally, and no source code is transmitted externally. The optional cloud-based explanation feature is only enabled when the user explicitly chooses to use it.

Before sending any data to external services, sensitive content is removed and reduced to minimal snippets. These snippets are intentionally small and do not include full files or confidential values. The system also avoids storing logs or intermediate data beyond what is required to generate the final report.

V. EXPERIMENTAL SETUP AND EVALUATION

A. Test Artifacts

To evaluate the system, a set of small Spring Boot applica-tions was created with intentionally introduced vulnerabilities. These test cases help demonstrate how the system behaves in controlled but realistic scenarios.

- **DemoApp-Auth:** Includes missing authentication on ac-tuator endpoints and weak password encoding.
- **DemoApp-DB:** Contains SQL queries built using string concatenation and unsafe deserialization.

These applications are not intended as full benchmarks, but rather as examples to validate detection and explanation workflows.

B. Evaluation Metrics

We used common evaluation metrics to see how well the system performed. These numbers show how well the system finds threats and how quickly it scans.

Precision: Measures how many of the detected vulnera-bilities are actually correct.

- **Recall:** Measures how many real vulnerabilities present in the system are successfully detected.
- **F1 Score:** Provides a balance between precision and recall.
- **Average Runtime:** Indicates the time taken to analyze each file.

These metrics are commonly used in vulnerability detection systems and help compare performance with existing tools.

C. Illustrative Results

The results shown below are based on small-scale experi-ments and are intended to demonstrate system behavior rather than provide a full benchmark comparison.

VI. RESULTS AND DISCUSSION

The evaluation of the proposed system highlights the effec-tiveness of combining static and dynamic analysis techniques. Static analysis enables rapid identification of common coding vulnerabilities and configuration issues across the application source code. Since it operates without executing the applica-tion, it provides early feedback during development and helps developers fix issues before deployment.

The proposed system generates an interactive dashboard summarizing the vulnerabilities detected during the analysis process.

However, static analysis alone may generate false positives due to lack of runtime context. Certain patterns detected in code may not always be exploitable in practice. To address this limitation, the dynamic analysis module performs runtime validation by interacting with application endpoints. This allows the system to verify whether identified vulnerabilities can actually be exploited.

Static analysis enables rapid identification of common coding vulnerabilities and configuration issues across the application source code.

Fig. 5: Dashboard summarizing detected vulnerabilities and severity distribution.

For example, endpoints detected during static analysis are tested using HTTP requests to determine if authentication is properly enforced. Similarly, configuration issues such as weak CORS policies and exposure of sensitive data are validated at runtime. This combination significantly improves the reliabil-ity of the results.

Fig. 6: Dashboard summarizing detected vulnerabilities and severity distribution.

Fig. 7: Dashboard summarizing detected vulnerabilities and severity distribution.

To address this limitation, the dynamic analysis module performs runtime validation by interacting with application endpoints Another important contribution of the system is the AI-based explanation module. Instead of presenting raw vulnerability data, the system generates simplified explana-tions along with actionable remediation steps. This improves usability, especially for students and developers who may not have strong security expertise.

The local-first design ensures that source code remains within the user's environment, addressing privacy concerns. While local models may provide shorter explanations, the optional cloud-based fallback improves explanation quality when required.

Overall, the system demonstrates a balanced approach that combines detection accuracy, runtime validation, and explain-ability. This makes it more practical compared to standalone static or dynamic tools.

A. Advantages of the Proposed System

The proposed system offers several advantages:

- Combines static and dynamic analysis for improved de-tection accuracy.
- Provides AI-generated explanations for better understand-ing of vulnerabilities.
- Ensures privacy through local-first execution.
- Uses modular detector design for easy extensibility.
- Suitable for both educational use and small-scale real-world applications.

The table shows that the proposed system achieves higher recall compared to reference tools, indicating that it is able to identify a larger number of vulnerabilities. However, the precision is slightly lower, which suggests the presence of some false positives due to rule-based detection.

Tool	Precision	Recall	F1	Avg Time (s)
Proposed Scanner	**0.55**	**1.00**	**0.71**	**0.45**
FindSecBugs	0.85	0.70	0.77	0.90
OWASP ZAP	0.70	0.60	0.64	2.50

Table I: Performance Evaluation

The average execution time is significantly lower than dynamic tools such as OWASP ZAP, as the system uses lightweight probes instead of full-scale scanning. This makes it suitable for quick analysis during development.

These results highlight the trade-off between detection accu-racy and performance, where the proposed system prioritizes coverage and usability.

B. Performance Analysis

The performance of the proposed system was evaluated in terms of detection capability and execution time. The results indicate that the system achieves high recall, meaning it is able to identify most of the vulnerabilities present in the test applications.

However, the precision is relatively lower compared to some established tools. This is expected due to the rule-based nature of the static analysis, which may flag patterns that are not always exploitable in practice.

The average runtime per file is lower compared to dynamic tools such as OWASP ZAP, as the system uses lightweight probes instead of full-scale scanning. This makes the system suitable for quick analysis during development.

Compared to tools like FindSecBugs, the proposed sys-tem offers faster execution and additional runtime validation through dynamic analysis. While FindSecBugs provides higher precision, it lacks runtime verification and explanation support. Overall, the system demonstrates a trade-off between speed, coverage, and precision. The integration of static and dynamic techniques ensures better coverage, while the lightweight design maintains efficiency.

C. Case Study (Detailed Analysis)

To better evaluate the system, a sample Spring Boot ap-plication containing multiple vulnerabilities was analyzed in detail.

The static analysis phase successfully detected issues such as SQL injection risks caused by string concatenation, hard-coded secrets in configuration files, and usage of weak cryp-tographic algorithms. These vulnerabilities were identified quickly without requiring execution of the application.

During dynamic analysis, the system interacted with appli-cation endpoints and identified runtime issues such as unpro-tected admin endpoints and overly permissive CORS config-urations. These checks confirmed whether the vulnerabilities detected during static analysis were actually exploitable. The

AI explanation module played a key role in improving us-ability. For each detected vulnerability, the system generated a simplified explanation along with recommended fixes. This helped in understanding the impact of vulnerabilities and how to resolve them. The combined approach of static detection, runtime validation, and AI explanation provided a comprehend sive view of the application's security posture. This case study demonstrates the effectiveness of the system in identifying and explaining vulnerabilities in real-world scenarios.

D. System Limitations

While the proposed system provides a practical approach to vulnerability detection, it has certain limitations.

The static analysis relies on predefined rules and pattern matching, which may not capture complex vulnerabilities involving deep data flow or multiple execution paths. Ad-vanced attacks that depend on application logic may remain undetected.

The dynamic analysis component is intentionally lightweight and does not perform deep penetration testing. As a result, certain runtime vulnerabilities may not be fully explored.

The AI explanation module depends on the capability of the underlying model. Local models may produce shorter explanations due to resource constraints, while cloud-based models introduce dependency on external services.

Another limitation is the lack of large-scale evaluation on real-world datasets. The current evaluation is based on small test applications designed to demonstrate system behavior.

Despite these limitations, the system provides a strong foundation for practical vulnerability detection and can be extended in future work.

E. Discussion

The proposed system highlights the importance of com-bining multiple analysis techniques to improve vulnerability detection. Static analysis alone is fast but may lack context, while dynamic analysis provides runtime validation but re-quires execution.

By integrating both approaches, the system achieves a balance between detection coverage and reliability. Static anal-ysis ensures early detection, while dynamic analysis confirms exploitability.

Another important observation is the role of explainability in security tools. Traditional scanners often produce outputs that are difficult to interpret. The AI explanation module improves usability by converting technical findings into understandable insights.

The system also demonstrates that privacy-aware design is feasible. By using local-first AI processing, sensitive source code remains within the user environment. This is particularly useful in academic and small-scale development settings.

Overall, the system shows that combining detection, vali-dation, and explanation leads to a more practical and user-friendly vulnerability scanning approach.

VII.DETECTORS VS SEVERITY

Detector Name	Severity	Vulnerability Type
Static Analysis (SAST)		
CrossSiteScriptingDetector	High	Reflected XSS
SqlConcatDetector	High	SQL Injection
HardcodedSecretDetector	High	Credential Leak
WeakHashingDetector	High	Cryptographic Weakness
JwtSecretDetector	High	Weak Authentication
CommandInjectionDetector	High	Remote Code Exec (RCE)
InsecureDeserializationDetector	High	Remote Code Exec (RCE)
XxeDetector	Medium	XML External Entity
OpenRedirectDetector	Medium	Phishing Risk
InsecureRandomnessDetector	Low	Predictable Tokens
Dynamic Analysis (DAST)		
SensitiveDataInResponseDetector	High	Information Disclosure
UnprotectedAdminEndpointDetector	High	Broken Access Control
HttpFuzzingDetector	Medium	Stability/Crash Risk
WeakCORSDetector	Medium	Misconfiguration

Table II: Implemented Detectors And Severity Mapping

VIII. CONCLUSION AND FUTURE WORK

In this work, a hybrid vulnerability scanning system for Spring Boot applications was presented, combining static analysis, lightweight dynamic testing, and AI-assisted explana-tions. This study introduced a hybrid vulnerability scanner for Spring Boot apps, using static checks, quick dynamic testing, and AI-generated explanations to help clarify what was found. The system was built to find security problems and make them simpler to understand and repair, particularly for students and smaller development teams.

The results suggest that static analysis helps spot everyday coding errors and setup problems fast. Meanwhile, dynamic analysis checks if these problems can really be used when the program runs. By using both methods together, the system cuts down on false alarms while still catching most common security weaknesses. This work mainly adds an AI explanation module. Rather than showing hard technical results, the system gives clear, easy explanations and suggests practical fixes. This helps developers use the tool even if they don't know much about application security.

With local-first design, your source code stays private, helping meet data security and compliance requirements. The system works well for common vulnerability patterns, but it still has its limits. Static, rule-based detectors can miss the tougher problems hidden in complex data flows or deep business logic. Likewise, the dynamic analysis part is built to be simple, so it might miss advanced runtime attacks.

How good an AI explanation is depends on the model; local versions may give shorter, less detailed answers than cloud-based ones. We can improve the system by making detection more accurate and easier to use. You can use advanced static analysis, like tracking data flow, to find deeper security weaknesses. You can expand the dynamic testing module with stronger tests, broader edge-case coverage, and more thorough ways to probe.

By upgrading local AI models or speeding up inference, you can make explanations clearer while keeping privacy safe. You can also connect security checks to CI/CD so scans run nonstop during development, and add IDE tools that show instant feedback as you write code. By adding support for other frameworks besides Spring Boot, the system can be used in more situations. Overall, it shows a simple, privacy-minded way to spot vulnerabilities, effective without being hard to use.

A. Real-World Applications

The proposed system can be applied in multiple real-world scenarios where security assessment is required during development.

In academic environments, the tool can help students un-derstand common vulnerabilities in Spring Boot applications. Instead of only identifying issues, it explains them in simple terms, making it easier for beginners to learn secure coding practices.

In small-scale development teams and startups, the system can be used as a lightweight security scanner integrated into the development workflow. Since it does not require complex setup or external infrastructure, it is suitable for rapid development environments.

The system can also be used during code reviews to identify potential vulnerabilities before deployment. Developers can quickly scan their applications and receive actionable feedback without relying on external tools.

Additionally, the privacy-first design makes it suitable for environments where source code confidentiality is important. Since analysis is performed locally, sensitive data is not exposed to external services.

These use cases demonstrate that the system is not limited to academic purposes but can also be applied in practical software development scenarios.

Tool	Static Analysis	Dynamic Analysis	AI Support
Proposed System	Yes	Yes	Yes
FindSecBugs	Yes	No	No
OWASP ZAP	No	Yes	No
Semgrep	Yes	No	No

Table III: Comparison of Proposed System with Existing Tools

B. Future Enhancements

Several improvements can be made to enhance the capabil-ities of the proposed system.

One important direction is the integration of advanced static analysis techniques such as data flow analysis and taint tracking. This would allow the system to detect more complex vulnerabilities that involve multiple execution paths.

The dynamic analysis module can be extended to include deeper runtime testing, including advanced fuzzing techniques and automated attack simulation. This would improve the sys-tem's ability to detect runtime vulnerabilities more accurately. Another area of improvement is the enhancement of the AI explanation module. Using more optimized local models or hybrid inference techniques can improve explanation quality while maintaining privacy.

Integration with CI/CD pipelines can enable continuous security scanning during development. This would allow vul-nerabilities to be detected and fixed earlier in the software lifecycle.

The system can also be extended to support other frame-works beyond Spring Boot, increasing its applicability across different types of web applications.

Finally, developing IDE plugins can provide real-time feed-back to developers while writing code, further improving usability and adoption.

ACKNOWLEDGMENT

We thank faculty and lab staff for guidance and infrastruc-ture support during this project.

REFERENCES

1. SonarSource, "SonarQube Documentation," 2025. [Online]. Available: https://docs.sonarsource.com/sonarqube/
2. FindSecBugs, "Find Security Bugs," 2025. [Online]. Available: https: //find-sec-bugs.github.io/
3. OWASP, "Zed Attack Proxy (ZAP) Documentation," 2025. [Online]. Available: https://www.zaproxy.org/docs/
4. Semgrep, "Semgrep Documentation," 2025. [Online]. Available: https: //semgrep.dev/docs/
5. Snyk, "Snyk Security Platform," 2025. [Online]. Available: https://docs. snyk.io/
6. OWASP Foundation, "OWASP Top 10: The Ten Most Critical Web Application Security Risks," 2021. [Online]. Available: https://owasp. org/www-project-top-ten/
7. Pivotal Software, "Spring Security Reference," 2025. [Online]. Avail-able: https://docs.spring.io/spring-security/reference/
8. Spring, "Spring Boot Documentation," 2025. [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/html/
9. Z. Li et al., "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," in *NDSS*, 2018.
10. Y. Zhou et al., "Devign: Effective Vulnerability Identification by Learn-ing Comprehensive Program Semantics," in *ASE*, 2019.
11. F. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *EMNLP*, 2020.
12. OWASP, "Automated Web Application Security Testing Using ZAP," 2024.
13. S. Neuhaus et al., "Predicting Vulnerable Software Components," in *ACM CCS*, 2007.
14. R. Seacord, "Secure Coding in C and C++," Addison-Wesley, 2013.
15. OpenAI, "Safety Best Practices," 2025. [Online]. Available: https:// platform.openai.com/docs/guides/safety-best-practices
16. OpenAI, "Data Usage Policies," 2025. [Online]. Available: https:// platform.openai.com/docs/guides/your-data
17. D. Stuttard and M. Pinto, "The Web Application Hacker's Handbook," Wiley, 2011.
18. Mozilla, "CORS Documentation," 2025. [Online]. Available: https:// developer.mozilla.org/en-US/docs/Web/HTTP/CORS
19. Oracle, "Java Security Overview," 2025. [Online]. Available: https: //docs.oracle.com/javase/
20. Docker, "Docker Security," 2025. [Online]. Available: https://docs. docker.com/engine/security/
21. Google, "API Security Best Practices," 2024.
22. NIST, "Security and Privacy Controls for Information Systems," 2020.
23. M. Sutton et al., "Fuzzing: Brute Force Vulnerability Discovery," 2007.
24. Oracle, "SecureRandom Class Documentation," 2025.
25. OWASP, "XML External Entity Prevention Cheat Sheet," 2023.