International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 10 | Oct - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

AI-Powered Code Review Assistant: A Research Paper

Prof. Saurabh Vyas[1], Dhammadeep Ganvir[2], Adharv S Aneesh[3], Harsha Bhoge[4], Sakshi Kubade[5], Utkarsh Shamkuwar[6]

Department of Computer Science and Engineering (Cyber Security)
G. H. Raisoni College of Engineering and Management, Nagpur (GHRCEMN), India

***

Abstract – Code security review is an important part of software development because vulnerable coding patterns can introduce risks into applications. Manual review can be time-consuming, while rule-based scanners may not provide sufficient contextual explanations for developers. This paper presents CodeGuard AI, an AI-assisted code security review platform designed to identify potentially vulnerable code, explain findings, suggest remediation, and support verification after changes. The project's described detection approach combines custom security rules and pattern matching with AI-generated explanations using Gemini. The workflow also includes code analysis through a web interface and support for scanning project files supplied in ZIP archives. Where implemented and enabled, post-fix verification can re-run analysis and tests to check whether a reported issue remains. The current project evidence described for this draft is limited to approximately 1–20 samples across multiple programming languages, with screenshots and scan records. Consequently, this paper treats the work as a preliminary implementation and does not claim statistically validated detection performance. Further evaluation using a larger, labelled benchmark and repeatable test protocol is required.

Key Words: CodeGuard AI, static code analysis, secure coding, vulnerability detection, AI-assisted code review, Gemini, pattern matching, remediation verification.

1. INTRODUCTION

Software vulnerabilities can arise from unsafe input handling, insecure API usage, weak validation, and other implementation mistakes. Developers need tools that identify suspicious code and communicate why it may be risky, where the problem occurs, and how it might be corrected. Traditional rule-based analysis is useful for repeatable checks, but its findings may require additional explanation and context. AI-generated explanations can make findings easier to understand, although they must be reviewed because generated recommendations can be incomplete or unsuitable for a particular codebase.

CodeGuard AI is intended to bring detection, explanation, remediation guidance, and verification into a single review workflow. The project context identifies custom rules and pattern matching as the detection foundation and Gemini as the AI component for explanations. The platform also includes a dashboard/code-editor workflow and ZIP-based project-file scanning. The research contribution should therefore be framed as an implementation and preliminary evaluation of an integrated AI-assisted code-review workflow, rather than as a new general-purpose autonomous cyber-defence system.

2. BACKGROUND OF CODE SECURITY THREATS

Software vulnerabilities can occur due to insecure coding practices, improper input validation, weak authentication, and misuse of security-sensitive functions. These weaknesses may allow attackers to access unauthorized information, manipulate application behaviour, or compromise system security. Identifying potential vulnerabilities during development is important for reducing security risks and improving software reliability.

Traditional security analysis tools use predefined rules and patterns to identify suspicious code. However, these tools may produce false positives or fail to identify vulnerabilities that depend on program context. AI-assisted code review can help explain security findings and suggest possible improvements, but its recommendations must be reviewed and tested before implementation. CodeGuard AI combines custom rules and pattern matching with AI-generated explanations to support a more understandable code-review process.

Common code security threats include:

1. SQL Injection: Occurs when unsafe user input changes the intended behaviour of a database query.
2. Command Injection: Occurs when untrusted input is used unsafely in operating-system commands.
3. Hard-Coded Credentials: Storing passwords, API keys, or other secrets directly in source code can expose sensitive information.
4. Improper Input Validation: Failing to validate input correctly can allow unexpected or potentially harmful data to affect an application.
5. Insecure API Usage: Using security-sensitive functions or APIs incorrectly may introduce weaknesses into software.

3. PROBLEM STATEMENT

Developers require practical tools that can identify potentially vulnerable source-code patterns and explain the findings in a form that supports corrective action. Existing approaches can create difficulties when security findings are difficult to understand, when scanners lack relevant rules, or when a proposed change is accepted without sufficient verification.

Manual code review can require substantial effort and may depend on specialist knowledge. Conventional rule-based tools can identify predefined patterns but may provide limited contextual explanation. AI-only analysis can generate readable feedback but may produce unsupported findings or recommendations that are not appropriate for the target application.

The research problem addressed by CodeGuard AI is therefore:
How can a web-based code-security review workflow combine custom rule-based detection and AI-generated explanations to make potential vulnerabilities more understandable and actionable, while maintaining developer oversight and supporting verification of changes?

4. OBJECTIVES OF THE STUDY

The main objectives of this research include:

1. Develop an integrated interface for submitting code for security analysis.
2. Identify potentially vulnerable patterns using the project's custom rules and pattern-matching engine.
3. Use Gemini to generate explanations or review guidance for detected findings, where configured.
4. Present remediation suggestions for developer review rather than treating AI-generated fixes as automatically correct.
5. Support project-file/ZIP scanning as described in the project implementation.
6. Provide a post-fix workflow that re-scans changed code and runs available tests, where implemented.
7. Record findings and supporting evidence so that behavior can be reviewed and evaluated.
8. Evaluate the prototype using a clearly labelled sample set and report only metrics supported by recorded evidence.

5. LITERATURE REVIEW

The application of static code analysis and Artificial Intelligence in software security has gained significant attention in recent years. Researchers have explored different techniques to identify software vulnerabilities, understand insecure coding practices, and recommend suitable corrections. Traditional static analysis tools examine source code using predefined rules and analysis techniques, whereas Large Language Models (LLMs) can interpret code context and generate natural-language explanations. These developments provide a foundation for AI-assisted code security review systems such as CodeGuard AI.

In **2023**, researchers investigated the ability of Large Language Models to identify and repair software vulnerabilities without specialized security training. The study *Examining Zero-Shot Vulnerability Repair with Large Language Models* evaluated the ability of general-purpose language models to generate security fixes under

different prompting conditions. The findings demonstrated that language models could produce useful fixes in simple and controlled scenarios; however, their performance was less dependable when dealing with real-world code. This research highlighted the importance of validating AI-generated corrections rather than accepting them without further review [1].

In **2024**, Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo conducted a systematic literature review titled *Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead*. Their study examined research on the use of LLMs for vulnerability detection and repair, including different models, adaptation techniques, and research challenges. The review identified opportunities for applying language models to software security while emphasizing the limitations of existing approaches. This research provides a useful foundation for systems that combine vulnerability identification with AI-generated explanations and remediation guidance [2].

In **2025**, Xin Zhou and his research team published an expanded review, *Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead*, in the *ACM Transactions on Software Engineering and Methodology*. The review examined 58 relevant research papers and organized existing approaches to LLM-based vulnerability detection and repair. It also discussed limitations and opportunities for future research. The study indicates that AI-based security analysis is an active research area, but further work is required to improve the reliability and practical applicability of vulnerability detection and repair methods [3].

In the same year, Damian Gnieciak and Tomasz Szandala presented *Large Language Models Versus Static Code Analysis Tools: A Systematic Benchmark for Vulnerability Detection*. Their study compared static analysis tools, including SonarQube, CodeQL, and Snyk Code, with several language models using vulnerable C# projects. The researchers evaluated detection performance, analysis time, and the effort required to review reported findings. Their results highlighted a trade-off between the contextual capabilities of language models and the more structured outputs of static analysis tools. This research supports investigating a hybrid approach in which rule-based detection is combined with AI-assisted interpretation, while recognizing that both approaches require evaluation [4].

In **2026**, Vladislav Belozerov, Peter J. Barclay, and Ashkan Sami investigated AI-assisted vulnerability detection and repair using code snippets from the DevGPT dataset. Their study examined 2,315 code snippets written in C, C++, and C#, identifying vulnerabilities through static scanners and manual inspection before evaluating several language models. The researchers found that the models could detect and repair many of the identified vulnerabilities, but they also highlighted the risk of confidently generated yet incorrect recommendations. Their findings reinforce the importance of combining automated analysis with manual review and re-scanning after proposed changes [5].

The reviewed studies demonstrate that static analysis provides a structured way to identify known risky coding patterns, while AI models can contribute contextual explanations and remediation suggestions. Nevertheless, challenges remain in handling complex code, controlling false positives, validating suggested fixes, and evaluating performance across programming languages. CodeGuard AI follows this research direction by combining custom rules and pattern matching with Gemini-assisted explanations and remediation guidance. Its intended workflow also includes project-file scanning and re-analysis after changes, where supported by the implementation. The project should therefore be evaluated using labelled vulnerable and non-vulnerable samples, recorded scan results, and verified test outcomes before making quantitative claims about detection accuracy or repair effectiveness.

6. PROPOSED CODEGUARD AI SYSTEM

CodeGuard AI is an AI-assisted code security review platform designed to help developers identify potential security vulnerabilities and improve code quality. The proposed system combines rule-based pattern matching and static code analysis with Artificial Intelligence to examine source code, explain detected issues in simple language, and suggest possible remediation steps. It also supports project-level scanning through ZIP uploads, maintains scan history, and allows developers to rescan code after applying corrections. The system aims to make code security review easier for students and beginner developers by presenting potential issues along with understandable explanations and recommendations. AI-generated suggestions are advisory and should be verified before being accepted.

6.1. System Overview

CodeGuard AI is described as a web-based platform for reviewing source code for potentially insecure patterns. Users can submit code through the interface or use the project's ZIP-based scanning workflow. The application processes the input, applies the configured detection logic, and presents findings. The AI component can provide explanatory text and remediation guidance associated with the review process.

The intended workflow consists of the following stages:

1. The user submits source code or a project archive.
2. The application receives and processes the submitted input.
3. The configured custom rules and pattern-matching engine examines eligible source code.
4. The application returns candidate findings and any available associated details.
5. Gemini generates explanations or review guidance where configured and available.
6. The developer reviews the finding and considers the proposed remediation.
7. The modified source can be submitted for another scan, and relevant tests can be run where supported.
8. The result can be compared with the original finding and recorded for later review where the implementation supports history.

6.2. User Interface and Code Submission

The user interface provides the entry point for code analysis. The project context describes a dashboard or code-editor workflow with an action for analysing code and a project-file verification flow.

A useful interface should make the relationship between submitted input and returned findings clear. Where supported, each finding should include its description, relevant file or line, severity, matched rule, and remediation guidance. The interface should also distinguish a detector-generated finding from an AI-generated explanation.

For reproducible evaluation, the user should be able to associate a scan result with a unique scan identifier or equivalent record. The actual fields displayed and stored must be documented from the implementation rather than assumed.

6.3. Input Processing and File Handling

Before analysis, the application needs to handle the submitted input according to the supported workflow. Pasted code and archive-based project input may require different processing steps.

Input processing should account for empty submissions, unsupported file types, malformed archives, unusually large inputs, and files that cannot be interpreted as source code. When scanning a project, the relationship between each finding and its source file should be preserved where possible.

ZIP-based scanning also introduces security requirements. Archive paths should be validated to prevent files from being written outside the intended extraction directory. File count, archive size, extracted size, and processing time may need limits to reduce resource-exhaustion risks. Unsupported or binary files should be handled predictably.

These are important implementation requirements, but the existence of ZIP scanning in the interface does not by itself prove that every safeguard is implemented. The final paper should report verified behaviour and identify any untested controls.

6.4. Custom Rules and Pattern-Matching Engine

The detection component is described as using custom security rules and pattern matching. Such rules identify source-code patterns associated with potentially unsafe programming practices. Each implemented rule should have a documented identifier, purpose, supported language, matching condition, assigned severity, and test cases. A rule may be tested with a positive example designed to trigger it and a negative example that resembles the risky pattern but uses a safe approach.

For instance, a project could evaluate a rule for unsafe query construction by comparing a vulnerable test fixture with a parameterized-query example. This is a proposed validation design, not evidence that CodeGuard AI currently contains or correctly implements such a rule. A reliable report must distinguish between a rule that exists in the source code and a vulnerability category that is

merely proposed for future testing. It should also distinguish between a pattern match and a confirmed exploitable vulnerability.

6.5. AI Explanation Component

Gemini is identified as the AI component in the supplied project documentation. Its described role is to provide explanatory review content and assistance with understanding findings. An effective explanation may describe the code pattern that triggered a warning, why it could create a security risk, under which conditions the risk may occur, and what the developer should inspect before making a change. A useful explanation should avoid presenting uncertain conclusions as established facts.

The implementation report should document the context supplied to the model, the format of the request, the output fields retained, and the handling of errors or unavailable responses. Where relevant, the model and prompt configuration should be recorded to support reproducibility. AI explanations should be assessed separately from detection. A correct explanation does not prove that the original finding was valid, and a successful API request does not prove that the generated content was accurate.

6.6. Remediation Suggestions

CodeGuard AI is intended to provide remediation suggestions that help developers address findings. Suggestions may include using safer APIs, validating untrusted input, separating data from executable commands, or changing a risky coding construct. The appropriate fix depends on the programming language, application framework, and surrounding logic. A recommendation that is safe in one context may be incomplete in another. For this reason, the developer should review the suggested change and ensure that it preserves the intended behaviour.

The paper should clearly state whether CodeGuard AI only displays suggestions or actually modifies source code. The available project notes support describing remediation guidance, but do not establish that every suggestion is automatically applied or validated. Therefore, the safest supported description is AI-assisted remediation guidance for developer review.

6.7. Post-Fix Verification

After a developer modifies code, the original analysis can be repeated to determine whether the targeted finding remains. Relevant unit tests, integration tests, or other available checks can provide additional evidence that the change has not broken required functionality.

A verification record should include the original finding, the modified source version, the result of the subsequent scan, the test command if applicable, the test exit status, and any new findings. The record should also state whether the targeted issue disappeared, remained, or could not be assessed.

A clean re-scan is useful but does not prove that the entire program is secure. The original detector may not cover all relevant weaknesses, and the modification may introduce an issue that the scanner cannot identify. Verification should therefore be described as evidence about a specific test and code version, not as a universal guarantee.

6.8. ZIP Project Scanning

The supplied project documentation identifies ZIP-project scanning as part of the workflow. Archive-based input can make the tool more convenient for users who want to review multiple files rather than submit a single code fragment. Project-level scanning requires decisions about supported extensions, ignored directories, archive size, file count, extraction behaviour, and the presentation of findings across multiple files. It is also important to identify whether findings preserve file and line information after processing.

These details should be checked against the actual implementation. The paper should not claim comprehensive project scanning or secure archive extraction unless these behaviours have been verified with appropriate tests.

6.9. Scan History and Result Management

The project context indicates that scan-history and refresh requests are part of the application workflow. History can help users revisit previous scans and help researchers preserve evidence of experimental runs.
For academic evaluation, each result should be associated with the input sample, application version, scan time, and test-case identifier where possible. If source code or AI output is stored, the paper should describe the storage behaviour and any relevant privacy considerations.

7. SYSTEM ARCHITECTURE

The system architecture of CodeGuard AI describes the structure and workflow of the proposed AI-assisted code security review platform. It explains how source code is submitted, processed, analyzed for potential security issues, reviewed using Artificial Intelligence, and presented to the user with possible remediation suggestions. The architecture consists of eight major layers that work together to support code analysis, project scanning, result storage, and post-fix verification.

Each layer performs a specific function in the overall system. The following subsections explain the purpose and working of each layer in two paragraphs, following the format of a research paper



7.1. Source Code Input

The Source Code Input stage is the entry point of the CodeGuard AI system. It allows users, such as developers and students, to submit source code for security analysis. Users can enter code directly into the application or upload a ZIP file containing a software project. This flexibility allows the system to support both individual code snippets and project-level analysis, depending on the available functionality.

After receiving the input, the system forwards the submitted code or project file for further processing. Direct code input is useful for testing small programs, while ZIP uploads can be useful when examining multiple source files. The submitted content provides the information required for subsequent analysis. Input validation and safe file handling are important to ensure that the system processes supported and appropriate files.

7.2. Input Processing (Validation, File Handling)

The Input Processing stage prepares the submitted source code for analysis. It checks and organizes the input so that the following stages can process it appropriately. For direct code submissions, this involves preparing the text for scanning. For ZIP uploads, processing may include identifying supported files, reading their contents, and organizing the project files for analysis.

This stage also plays an important role in handling invalid or unsupported inputs. For example, the system may need to handle empty submissions, unsupported file formats, or corrupted ZIP archives. For project uploads, safe extraction and appropriate file-size limits help reduce security risks. Proper input processing improves the consistency of the scanning workflow and helps prevent unsuitable input from disrupting the application.

7.3. Static Code Analysis (Rules, Pattern Matching)

The Static Code Analysis stage examines the submitted source code using predefined rules and pattern-matching techniques. It searches for code patterns that may indicate security weaknesses or unsafe programming practices. Unlike dynamic testing, static analysis can inspect source code without executing the program. This makes it useful for identifying potential issues early in the software development process.

When a code pattern matches an available detection rule, the system can generate a finding describing the potential issue and its location in the source code. These findings form the initial technical results of the review. The quality of the analysis depends on the coverage and accuracy of the implemented rules. A detected pattern does not always prove that a vulnerability is exploitable, so findings should be reviewed before conclusions are made.

7.4. AI-Based Review (Gemini Explanations)

The AI-Based Review stage uses the Gemini AI model to provide explanations for findings identified during static code analysis. Rule-based detection can identify suspicious patterns, but users may need additional information to understand why a pattern matters. The AI model can explain the possible security implications of a finding in simple language and provide context that helps users understand the reported issue.

This stage can also help users understand the general principles of secure programming associated with a finding. For example, an explanation may describe why untrusted input requires careful handling or why a particular coding practice could create a security risk. The quality of the explanation depends on the supplied code context and the model's response. Since AI-generated explanations can be incomplete or incorrect, they should be treated as assistance rather than definitive security judgments.

7.5. Remediation Suggestions (Recommended Fixes)

The Remediation Suggestions stage provides possible ways to address the issues identified during code analysis. Based on the detection results and AI-generated explanations, the system can recommend coding changes or safer programming practices. These recommendations are intended to help developers understand how a reported issue might be corrected and how similar problems could be avoided in future development.

The suggested changes should be examined before they are applied to the original project. A recommendation may not account for every dependency, business requirement, or surrounding code condition. Therefore, developers should review the proposed solution and make any necessary modifications. Remediation suggestions are advisory; they should not be described as automatically applied fixes unless the implementation actually modifies the source code.

7.6. Post-Fix Verification (Re-scan, Available Tests)

The Post-Fix Verification stage allows users to examine the source code again after making corrections. The modified code can be submitted for another scan, and the new findings can be compared with the original results. This helps determine whether the previously reported pattern is still detected and whether additional issues appear after the changes.

Where suitable automated tests are available, they can provide further evidence about the behavior of the modified program. However, the disappearance of a finding alone does not guarantee that the vulnerability has been completely eliminated. The verification result should reflect the checks that were actually performed, such as re-scanning or running available tests. This stage supports a more careful review of proposed corrections and encourages developers to validate their changes.

7.7. Scan History and Reporting Dashboard

The Scan History and Reporting Dashboard stage presents scan results in an organized format and supports access to previous analyses. Depending on the implemented features, the dashboard may display findings, scan status, timestamps, and other information associated with each scan. Persistent scan records help users revisit earlier results without having to repeat the same analysis.

The history feature can also help users compare successive scans and observe changes in the reported findings after code modifications. This is useful for maintaining a record of the code review process and supporting project demonstrations or development reviews. Appropriate access control and secure storage should be considered when scan records contain source code or sensitive project information. The dashboard should clearly distinguish detected issues, AI-generated recommendations, and verification outcomes.

8. IMPLEMENTATION METHODOLOGY

The implementation methodology of CodeGuard AI describes the step-by-step process used to design, develop, integrate, and evaluate the proposed AI-assisted code security review platform. The system combines static code analysis, rule-based pattern matching, Gemini-based explanations, remediation suggestions, and post-fix verification to help users identify and understand potential security issues in source code. The implementation process is divided into ten major stages, beginning with requirement analysis and system design and progressing through code processing, vulnerability detection, AI integration, result management, testing, and deployment. Each stage has a specific responsibility within the overall workflow, ensuring that the system

follows a structured approach to code security review.real-time threat detection, and automated response mechanisms.

8.1. Requirement Analysis

Requirement analysis is the initial stage of the implementation process. It involves identifying the problems the system is intended to address and defining the functional and non-functional requirements of CodeGuard AI. The main objective is to develop a platform that assists developers and students in reviewing source code, identifying potential security weaknesses, understanding detected issues, and receiving recommendations for improvement.

The functional requirements define the main operations that the system should perform, while non-functional requirements address qualities such as usability, reliability, security, and maintainability.

The main requirements include:

- **Source Code Submission:** The system should allow users to submit source code for analysis through the available input interface.
- **Project Upload:** ZIP-based project submission can support the analysis of multiple source files, provided the upload functionality is implemented.
- **Security Analysis:** The system should examine source code using predefined rules and pattern-matching techniques.
- **AI-Based Explanation:** The Gemini model should explain detected findings in understandable language.
- **Remediation Suggestions:** The system should provide possible corrections and secure coding recommendations.
- **Result Management:** Users should be able to view scan results and access previous records if history functionality is available.
- **Verification:** Users should be able to rescan modified code to review changes in the reported findings.

8.2. System Design and Architecture

System design defines the structure of CodeGuard AI and the relationship between its components. The platform is organized into separate functional layers, including the user interface, API and scan orchestration, input processing, static code analysis, AI-based review, remediation suggestions, post-fix verification, and scan history and reporting. This modular structure helps separate user interaction, code processing, detection, AI assistance, and result presentation.

The architecture also defines the flow of information between the components. When a user submits source code, the system prepares the input and sends it to the detection component. The resulting findings can then be passed to the AI component for additional explanation and recommendations. The application subsequently presents the available results and stores scan information where supported. Separating these responsibilities can improve maintainability and make individual components easier to test.

The major design considerations include:

- **Modular Design:** Each component performs a defined task within the code-review workflow.
- **Data Flow:** The architecture establishes how code, findings, AI responses, and scan records move through the system.
- **Component Integration:** The frontend, backend, analysis engine, AI service, and storage layer must communicate correctly.
- **Error Handling:** The system should handle invalid inputs and failures in individual services without presenting incomplete operations as successful.
- **Security Considerations:** Source code, uploaded archives, API credentials, and stored results should be handled securely.

8.3. Source Code Input and Preprocessing

The source-code input stage receives the program or project that the user wants to analyze. CodeGuard AI is designed to support direct source-code input and project submission through ZIP files. Direct input is useful when

a developer wants to examine a small code fragment, whereas project uploads can support the analysis of multiple source files in one workflow.

Before analysis begins, the submitted input must be prepared for the detection process. This may involve checking whether the input is empty, identifying supported file types, reading source files, and organizing the submitted content. For ZIP-based uploads, the system should also handle corrupted archives, unsupported files, and file-size restrictions appropriately.

The main preprocessing operations include:

- **Input Validation:** Check that the submitted content is present and suitable for processing.
- **File Identification:** Identify the source files that the system is configured to support.
- **File Reading:** Read the contents of supported source files for further analysis.
- **Archive Handling:** Process ZIP uploads and organize their contents if project scanning is implemented.
- **Error Management:** Report invalid files or unsupported input formats clearly.
- **Safe Processing:** Apply appropriate extraction controls to reduce the risk of path traversal and resource-exhaustion attacks.

8.4. Static Code Analysis and Rule-Based Detection

Static code analysis is one of the central components of CodeGuard AI. It examines source code without executing the program, using predefined security rules and pattern-matching techniques to identify code patterns that may indicate weaknesses. This approach can help developers detect potentially problematic code during development, before the program is deployed.

The detection process compares sections of the submitted code against the rules implemented in the analysis engine. When a pattern matches a rule, the system can generate a finding containing information about the possible issue and its location. The resulting findings provide the technical basis for the subsequent AI-based review.

The actual detection capabilities depend on the rules, programming languages, and file types supported by the implementation.

The main detection operations include:

- **Rule Definition:** Establish patterns that represent potentially insecure coding practices.
- **Pattern Matching:** Compare submitted code against the implemented rules.
- **Finding Generation:** Record information about patterns that match the rules.
- **Location Identification:** Identify the affected file or code location where supported.
- **Severity Classification:** Assign severity levels if this functionality is implemented.
- **Result Preparation:** Organize the findings so that they can be displayed or passed to the AI review component.

Static analysis has limitations. A matching pattern does not necessarily prove that a vulnerability is exploitable, and a rule-based scanner may miss problems that are not covered by its rules. Therefore, the findings should be interpreted as potential issues that may require further review.

8.5. AI-Based Review Using Gemini

The AI-based review stage uses the Gemini model to provide additional explanations for findings produced by the detection engine. Static analysis can identify patterns associated with potential security problems, but the results may not be sufficiently clear to users who are unfamiliar with the underlying vulnerability. The AI component aims to bridge this gap by explaining the reported issue in understandable language and describing its possible security implications.

The system can provide the relevant finding and available code context to the AI service and use the returned response to assist the user. Depending on the model's response and the implementation, the explanation may describe the issue, discuss its possible impact, and suggest safer coding practices. The AI response should remain distinguishable from the original scanner finding because it is generated by a language

model rather than directly established by the detection rule.

The main AI review operations include:

- **Finding Context:** Prepare the relevant finding and code information for AI review.
- **AI Request:** Send the appropriate information to the configured Gemini service.
- **Issue Explanation:** Generate a natural-language explanation of the potential problem.
- **Security Context:** Describe the possible impact of the reported coding pattern.
- **Response Processing:** Receive and present the generated explanation in the application.
- **Error Handling:** Handle unavailable services, timeouts, or incomplete model responses where appropriate.

AI-generated explanations are not guaranteed to be correct. They should be reviewed against the source code and the original detection evidence before being relied upon. Sensitive source code should also be handled carefully when sent to an external AI service.

8.6. Remediation Suggestion Generation

The remediation stage focuses on helping users understand how a reported issue might be addressed. After a potential security problem has been identified and explained, the system can provide recommendations for safer coding practices or possible modifications to the affected code. These suggestions are intended to support developers in understanding corrective actions rather than simply displaying a warning.
The recommendations may be based on the detected issue, the available source-code context, and the AI-generated explanation. The developer should examine the suggested approach before applying it because a correction must be appropriate for the application's requirements and surrounding implementation. A recommendation that appears correct in isolation may not be suitable for every project.

The main remediation operations include:

- **Issue-Based Guidance:** Associate recommendations with the reported finding.
- **Suggested Correction:** Provide a possible approach to address the issue.
- **Secure Coding Practices:** Explain safer alternatives where appropriate.
- **Developer Review:** Allow the user to assess whether the recommendation suits the project.
- **Code Modification:** Apply changes manually unless automatic patching is explicitly implemented.
- **Re-Scanning:** Submit the modified code for another analysis to review the resulting findings.

8.7. Post-Fix Verification and Re-Scanning

Post-fix verification is the stage in which modified code is examined after a developer has attempted to correct a reported issue. The purpose is to determine whether the new scan produces different findings and whether the previously detected pattern remains present. This supports an iterative review process in which developers can modify their code, analyze it again, and inspect the new results.

The verification process can compare the original scan with the subsequent scan. If the original finding is no longer reported, the system may indicate that the corresponding pattern was not detected during the new scan. However, this result alone does not prove that the vulnerability has been completely removed. Where suitable automated tests are available, they can provide additional evidence about the revised program's behavior.

The main verification operations include:

- **Modified Code Submission:** Receive the revised source code for another scan.
- **Repeated Analysis:** Run the available static analysis rules against the modified code.
- **Finding Comparison:** Compare the new results with the earlier findings when comparison functionality is available.
- **Remaining Issue Identification:** Check whether the original pattern or other findings are still reported.

- **Test Execution:** Run available automated tests if the system supports this operation.
- **Verification Reporting:** Present the observed results without overstating what the checks prove.

8.8. Scan History and Result Management

The scan-history component manages information generated during the code-review process. When a scan is completed, the application may retain the result and associated metadata so that the user can retrieve it later. Depending on the implemented storage structure, this information may include a scan identifier, timestamp, findings, processing status, and other relevant details.

The reporting interface can display the current findings and provide access to earlier scan records. This supports repeated testing and allows users to review changes across multiple scans. The implementation should also consider data confidentiality because scan records may contain sensitive source-code information or details about software vulnerabilities.

The main result-management operations include:

- **Result Storage:** Save scan information when persistent storage is implemented.
- **Record Identification:** Associate results with a particular scan.
- **History Retrieval:** Retrieve previously stored results for display.
- **Result Presentation:** Display findings and available explanations in an organized format.
- **Scan Comparison:** Support comparison between earlier and later scans if this feature exists.
- **Data Protection:** Apply suitable access controls and retention practices to stored records.

8.9. Testing and Evaluation

Testing and evaluation are necessary to determine whether CodeGuard AI performs its intended functions. Functional testing checks that the different components operate correctly, including code submission, input processing, static analysis, AI-service communication, result presentation, and scan-history retrieval. Testing should also cover invalid inputs and error conditions to determine how the application responds when an operation cannot be completed.

Security-detection evaluation requires a separate set of test cases with known expected outcomes. Each test should record the submitted code, the expected result, the actual result, and whether the detection was correct. If remediation is being evaluated, the modified code should be examined again, and appropriate tests should be performed where available. Quantitative metrics should be calculated from the recorded evidence rather than estimated.

8.10. Deployment and Maintenance

Deployment involves preparing CodeGuard AI for use in a local environment, demonstration, or other intended setting. The process includes configuring the application, installing the required dependencies, setting up the backend and frontend, and verifying communication between the components. If the system uses the Gemini service, the relevant API configuration and connectivity must also be checked.

After deployment, maintenance is required to keep the application reliable and secure. This may include updating dependencies, correcting implementation defects, improving detection rules, and extending support for additional file types or programming languages. The system should also be tested after significant changes to ensure that previously working features continue to operate correctly.

The main deployment and maintenance activities include:

- **Environment Configuration:** Prepare the runtime environment and required dependencies.
- **Service Configuration:** Configure the backend, frontend, and AI integration as applicable.
- **Credential Protection:** Store API credentials securely and avoid exposing them in source code or public repositories.
- **Deployment Testing:** Verify that the deployed application performs its intended functions.

- **Dependency Maintenance:** Update dependencies and address known security issues.
- **Rule Improvement:** Refine detection rules based on test results and identified limitations.
- **Future Enhancement:** Extend functionality based on evaluation findings and user requirements.

8.11. Overall Implementation Workflow

The implementation methodology follows a sequence of connected stages, beginning with requirement analysis and system design. Source code is then submitted and prepared for analysis, after which the detection engine identifies potential issues. The AI component provides explanations and remediation suggestions, while re-scanning supports review of modified code. Results are organized for presentation and history management, and the complete application is evaluated through functional, integration, and security-detection testing.

9. LIMITATIONS AND THREATS TO VALIDITY

9.1. Limited Sample Size

The supplied project context describes a small sample set of approximately 1–20 samples. Such a sample is insufficient by itself to demonstrate general performance across programming languages, frameworks, and vulnerability categories.

9.2. Incomplete Ground Truth

Detection evaluation requires a trusted expected outcome for each test case. Without labelled cases and independent review, it is difficult to determine whether a finding is correct or whether an unreported issue represents a false negative.

9.3. Rule and Language Coverage

The exact rule catalogue, supported language mapping, severity assignment, and matching conditions must be confirmed from the source code. A user interface that accepts code in a language does not prove that the detection engine contains effective security rules for that language.

9.4. AI Output Reliability

AI-generated explanations and remediation suggestions can be inaccurate, incomplete, or incompatible with the application's requirements. They may also change with model versions, prompts, or configuration. The AI component should be assessed independently and its outputs should not be treated as ground truth.

9.5. Remediation Verification

A proposed correction does not demonstrate that a vulnerability has been resolved. A re-scan may fail to detect the original issue for reasons unrelated to the change, and a fix may introduce a new problem. Verification should include source inspection and relevant tests where possible.

9.6. ZIP Processing and Resource Safety

Archive-based scanning creates risks involving malformed archives, path traversal, large extracted content, unsupported files, and excessive resource use. The supplied project notes identify ZIP scanning, but they do not establish that every possible safeguard has been implemented and tested.

9.7. Limited Comparative Evaluation

No complete comparison against a suitable baseline scanner is established in the supplied evidence. Therefore, the paper should not claim superiority over existing tools. A fair comparison requires identical test cases, documented configurations, and consistent evaluation criteria.

9.8. Reproducibility and Data Privacy

Reproducibility may be affected by changes to application versions, rules, model configurations, or prompts. In addition, sending source code to an external AI service can create privacy concerns. The implementation and publication should document relevant data handling, avoid publishing sensitive source code, and use appropriate dummy values in test examples.

Future research can focus on enhancing the accuracy, scalability, and adaptability of AI-driven cyber defence systems. With the rapid evolution of cyber threats, traditional machine learning models may not always

sufficient to detect highly sophisticated attacks. Therefore, advanced approaches such as deep neural networks and reinforcement learning can be explored to develop more intelligent and adaptive security mechanisms. These techniques can enable the system to learn complex attack patterns, improve anomaly detection capabilities, and reduce false positive rates. Additionally, incorporating continuous learning frameworks can allow the defence system to update its knowledge automatically as new types of cyber threats emerge.

Another important direction for future research involves the integration of AI-based cyber defence systems with emerging technologies such as cloud computing and Internet of Things (IoT) environments. As modern digital infrastructures increasingly rely on interconnected devices and distributed networks, securing these systems becomes more challenging. Future work may focus on designing lightweight and scalable security models that can protect IoT devices and cloud platforms in real time. Furthermore, combining AI-based threat intelligence with distributed security architectures may significantly enhance the resilience of modern networks against large-scale cyber attacks.

10. FUTURE SCOPE

10.1. Expansion of the Test Dataset

Future work should create a larger and more balanced dataset covering supported programming languages, vulnerability categories, vulnerable examples, and safe counterexamples. Each case should have a documented expected result and rationale. The dataset should include both simple patterns and examples requiring more contextual analysis.

10.2. Improved Detection Coverage

The custom-rule engine can be expanded after identifying the categories most relevant to the supported languages. Each new rule should be accompanied by positive and negative regression tests. The rule inventory should record the matching condition, severity, supported language, and known limitations.

10.3. Evaluation of AI Explanations

Future experiments should assess whether explanations are accurate, understandable, and useful to developers. Reviewers can score outputs using a predefined rubric. Where practical, multiple reviewers should evaluate a subset of the same explanations to examine consistency.

10.4. Validation of Remediation Suggestions

Remediation guidance should be evaluated using before-and-after code examples. The researcher should check whether the proposed change addresses the targeted issue, preserves required functionality, and passes relevant tests. Security-sensitive examples may require manual expert review.

10.5. Automated Regression Testing

A regression test suite can help ensure that existing rules continue to behave as expected after code changes. Each rule should have positive examples that should be detected and negative examples that should not be flagged. Repeated execution can reveal unintended changes in behaviour.

10.6. Secure ZIP Project Processing

Future development can strengthen archive validation, path checks, file-size limits, file-count limits, and handling of unsupported or malformed files. Tests should verify that archive processing remains within the intended working directory and does not consume excessive resources.

10.7. Baseline Comparison

CodeGuard AI can be compared with suitable established static-analysis or security-scanning tools. The comparison should use the same labelled samples and documented configurations. Metrics should include detection outcomes, false positives, analysis time, and other relevant measurements. Differences should be interpreted in light of the tools' distinct capabilities and intended use.

10.8. Improved Reporting and Traceability

Future versions can improve research reproducibility by recording the application version, rule version, scan identifier, test-case identifier, model configuration, and relevant outputs. Reports should distinguish confirmed observations, AI-generated guidance, and verification outcomes.

11. CONCLUSION

1) CodeGuard AI combines static code analysis and Artificial Intelligence to identify potential security issues in source code. It provides a structured approach to reviewing code security.

2) The system uses predefined rules and pattern matching to detect potentially insecure coding patterns. This helps developers recognize possible vulnerabilities during software development.

3) Gemini AI explains detected issues in simple and understandable language. This helps users understand the causes and possible security impacts of reported findings.

4) The platform provides remediation suggestions to help developers improve their source code. These recommendations encourage the use of safer coding practices.

5) CodeGuard AI supports source-code submission for security analysis. This allows users to examine individual code snippets and review potential security weaknesses.

6) ZIP-based project scanning can help analyze multiple source files within a software project. This provides a convenient approach to reviewing project code, depending on the supported implementation.

7) The post-fix verification process allows users to rescan modified source code. The new results help users check whether previously detected patterns remain after corrections.

8) Scan history and reporting help users organize and revisit previous analysis results. This supports the review of findings across multiple scanning sessions.

9) CodeGuard AI can assist students and beginner developers in learning about software vulnerabilities and secure coding practices. It connects practical code analysis with AI-generated explanations.

10) The system can be improved by expanding detection rules, supporting more programming languages, and strengthening verification methods. Further testing is required to evaluate its detection capabilities and the reliability of its recommendations.

.

12. REFERENCES

[1] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining Zero-Shot Vulnerability Repair with Large Language Models," Proceedings of the IEEE Symposium on Security and Privacy (SP), 2023. DOI: 10.1109/SP46215.2023.10179324

[2] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead," arXiv preprint arXiv:2404.02525, 2024. Available: DOI : https://arxiv.org/abs/2404.02525.

[3] D. de Fitero-Dominguez, E. Garcia-Lopez, A. Garcia-Cabot, and J.-J. Martinez-Herraiz, "Enhanced Automated Code Vulnerability Repair Using Large Language Models," Engineering Applications of Artificial Intelligence, vol. 138, Part A, Article 109291, 2024. DOI: 10.1016/j.engappai.2024.109291.

[4] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead," ACM Transactions on Software Engineering and Methodology, vol. 34, no. 5, Article 145, pp. 1–31, 2025. DOI: 10.1145/3708522.

[5] D. Gnieciak and T. Szandala, "Large Language Models Versus Static Code Analysis Tools: A Systematic Benchmark for Vulnerability Detection," arXiv preprint arXiv:2508.04448, 2025. Available DOI: https://arxiv.org/abs/2508.04448.

[6] V. Belozerov, P. J. Barclay, and A. Sami, "Secure Coding with AI – From Detection to Repair," Empirical Software Engineering, vol. 31, Article 93, 2026. DOI: 10.1007/s10664-026-10812-8