iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** IX **Month of publication:** September 2026

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IX Sep 2026- Available at www.ijraset.com

Malice-Scan: A Configurable Multi-Language Static Security Scanner for Pattern-Based Code Analysis

Pilli Chandana Das[1], D Venkateswarlu[2]
CSE dept., Amrita Sai Institute of Science and Technology, Vijayawada, Andra Pradesh, India

Abstract: ***Software projects often contain insecure coding practices, suspicious constructs, exposed secrets, and other patterns that require early identification during development and code review. This paper presents Malice-Scan, a configurable static analysis security scanner designed to inspect source code without executing it. The implemented system supports JavaScript, Python, PHP, Java, Ruby, and Shell scripting. Its architecture separates command-line orchestration, the detection engine, configuration management, utility functions, and multi-format reporting. The scanner applies forty configurable regular-expression-based detection patterns covering code injection, backdoors and network operations, runtime-error patterns, cryptographic weaknesses, and other security-relevant issues.***

It further incorporates heuristic checks for minified JavaScript and large Base64-like content, along with confidence estimation based on severity, dangerous keywords, entropy, and contextual indicators. Practical safeguards include file-size limits, regular-expression timeouts, Base64 decoding limits, scan-depth limits, path-traversal controls, and per-rule finding limits. Findings can be filtered by severity and exported in JSON, HTML, or CSV formats. The test structure includes vulnerable and clean samples, with a comprehensive test project containing 77 intentional issues across six supported languages. The resulting system provides a lightweight and extensible approach for security-oriented source-code scanning and can be integrated into local review and CI/CD workflows.

Keywords: Static Application Security Testing; Static Code Analysis; Secure Software Development; Vulnerability Detection; Regular Expressions; Multi-Language Scanner; Software Security; CI/CD.

I. INTRODUCTION

Security defects and suspicious code constructs can be introduced during routine software development and may remain unnoticed until code review, deployment, or incident response. Static analysis is therefore an important complementary technique because it allows source code to be inspected before execution. Prior work has highlighted the role of static analysis in identifying security-relevant coding defects [1], while secure code review guidance also recognizes the practical value of automated source-code scanning for locating recurring patterns across large codebases [2].

However, static scanners are not substitutes for expert review. Pattern-based approaches can efficiently locate known syntactic structures but may lack sufficient semantic and business context, making false positives and coverage limitations important concerns [2], [3]. Recent empirical work on configurable SAST rules likewise shows that rule-set coverage materially affects what vulnerabilities a scanner can identify [4].

Malice-Scan was developed as a practical, configurable static security scanner for multi-language source projects. The project emphasizes pattern-driven detection, lightweight heuristics, configurable scanning behavior, confidence estimation, and actionable reporting. The supported languages are JavaScript, Python, PHP, Java, Ruby, and Shell scripting. The system is intended for security teams, DevOps engineers, code reviewers, development teams, and other users requiring repeatable source-code inspection.

The main contributions of the implemented project are: (i) a modular static scanning architecture with command-line orchestration and a dedicated detection engine; (ii) forty configurable detection patterns organized across multiple security and reliability categories; (iii) heuristic checks for selected obfuscation-related indicators; (iv) configurable safety controls for scanning and regular-expression execution; and (v) JSON, HTML, and CSV reporting suitable for different review workflows. The paper describes the implemented system and does not claim universal vulnerability coverage or superior detection accuracy.

II. LITERATURE REVIEW / RELATED WORK

Chess and McGraw [1] describe static analysis as an important mechanism for identifying security defects during software development. Their work provides foundational context for examining code without executing it and supports the broader use of automated analysis as part of secure software practices.

OWASP guidance distinguishes simple pattern-search scanners from more sophisticated static analyzers and notes that automated source-code scanners are particularly useful for repeatedly locating known insecure patterns and reporting their locations. At the same time, the guidance emphasizes limitations involving business logic, design issues, restricted rule scope, and false positives [2]. These limitations are directly relevant to a regex-oriented implementation such as Malice-Scan and motivate transparent reporting rather than unsupported claims of complete vulnerability detection.

Studies comparing static analysis tools also show that detection capability can vary by weakness category and tool design. A Java-focused case study comparing multiple open-source tools reported that particular weaknesses were detected differently across tools [5]. This reinforces the importance of matching a scanner's rule set to the intended vulnerability categories.

More recent research has investigated the effect of rule coverage on SAST effectiveness. Bennett et al. [4] demonstrate that missing vulnerable patterns can substantially limit detection and that rule augmentation can improve coverage for the evaluated cases. This supports the extensible pattern-file design used by Malice-Scan, where additional rules can be introduced without redesigning the overall scanning workflow.

The proposed system does not aim to replace semantic analyzers or advanced SAST platforms. Instead, Malice-Scan provides a lightweight, multi-language, configurable scanner centered on explicit patterns, selected heuristics, and portable report formats. Its role is to assist early security review and triage.

III. PROPOSED SYSTEM

A. *Design Overview*

The proposed system accepts a target file or directory and scans supported source files recursively. The workflow initializes configuration and detection patterns, discovers eligible files, applies exclusion rules, determines language from file extension, reads text content subject to configured limits, performs pattern matching and heuristic checks, aggregates findings, applies severity filtering, and generates the requested report.

B. *Detection Coverage*

The implemented rule set contains forty patterns, identified as R001 to R040. The rules cover four broad groups: code injection and dangerous functions; backdoors and network operations; runtime-error patterns; and cryptography and security issues. Examples include hardcoded secrets, dangerous eval/exec usage, command and SQL injection patterns, PHP webshell and reverse-shell indicators, Base64 payloads, unsafe deserialization, path traversal, weak cryptographic algorithms, insecure random number generation, SSL/TLS weaknesses, insecure cookie flags, CORS misconfigurations, SSRF patterns, and JavaScript prototype pollution. The project treats these as detection patterns rather than proof that every matched instance is an exploitable vulnerability.

C. *Configurability and Reporting*

Scanning behavior is controlled through configuration settings, including supported extensions, context lines, maximum file size, Base64 decoding limits, regular-expression timeout, heuristic confidence values, reporting limits, path-traversal controls, maximum scan depth, and logging settings. Findings can be reported in JSON, HTML, and CSV. The HTML format includes an executive summary, severity-oriented presentation, filtering and search capabilities, code snippets, and tags; the CSV format supports spreadsheet-oriented analysis; and the JSON format preserves structured metadata and findings for downstream processing.

IV. SYSTEM ARCHITECTURE AND METHODOLOGY

A. *Layered Architecture*

Malice-Scan follows a layered architecture. The CLI layer, implemented in scan.py, handles argument parsing, user interaction, output-format selection, logging setup, and workflow orchestration. The detection engine, implemented in detector.py, performs file scanning, language-specific pattern matching, heuristic analysis, and confidence estimation. Configuration and patterns are supplied through the configuration manager and JSON pattern definitions, while utility functions provide entropy calculation, Base64 checks, minified-JavaScript detection, snippet extraction, file-size formatting, and text-file checks. The reporter layer generates JSON, HTML, and CSV output and prints scan summaries.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IX Sep 2026- Available at www.ijraset.com

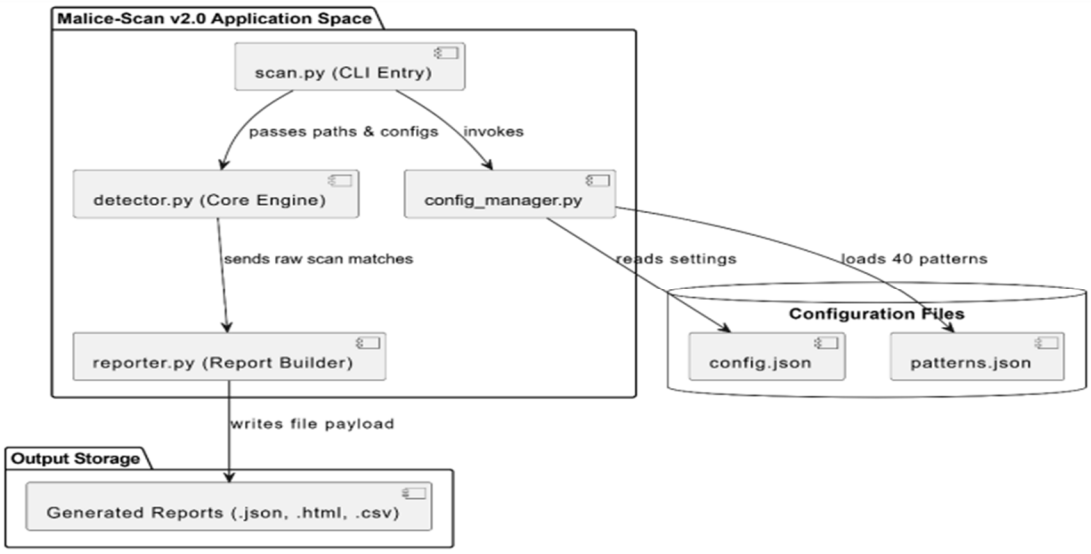


Figure 1. Architecture of Malice-Scan.

B. Scanning Methodology

The scanning process begins when the user supplies a target path and optional configuration parameters. The system loads configuration and patterns, recursively discovers candidate files, and applies exclusion patterns. For each eligible file, the language is inferred from the extension, configured size restrictions are checked, and the text content is read. Language-applicable patterns are then selected and evaluated with timeout protection. For each match, the scanner records the line number, surrounding code snippet, rule metadata, severity, confidence estimate, and tags. Base64-related context is checked near relevant matches, followed by file-level heuristics for minified JavaScript and large Base64-like blobs. Findings are aggregated and optionally filtered by minimum severity before reporting.

Figure 2. End-to-end scanning workflow.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IX Sep 2026- Available at www.ijraset.com

C. Confidence Estimation and Heuristics

The confidence score ranges from 0.0 to 1.0 and is influenced by the base severity level, dangerous keywords, file entropy, and contextual indicators such as whether the match appears in a comment. The project also uses file-level heuristics to flag minified JavaScript and large Base64 blobs. Shannon entropy is used as an indicator of high-entropy content associated with encoding, encryption, or obfuscation. Base64 checks use structural heuristics such as minimum length, character alphabet, and padding validity. These mechanisms support prioritization but are not presented as statistically calibrated vulnerability probabilities.

TABLE I. MAJOR IMPLEMENTED COMPONENTS

Component	Primary Responsibility
scan.py	CLI argument handling, logging, format validation, severity filtering, workflow orchestration
detector.py	Folder and file scanning, pattern matching, heuristic checks, confidence estimation
config_manager.py	Loading, merging, retrieving, setting, and saving configuration
reporter.py	JSON, HTML, and CSV report generation; console summary
utils.py	Entropy, Base64, minified-JS, snippets, text-file checks, formatting
patterns.json	Rule definitions containing ID, name, description, languages, severity, regex, and tags

V. IMPLEMENTATION

A. Command-Line Interface

The CLI accepts a target path and supports options for output location, output format, custom pattern and configuration files, minimum severity, repeated exclusion patterns, verbose mode, quiet mode, and version display. The scanner returns exit code 0 when no findings are detected and exit code 1 when findings are detected or an error occurs, enabling basic automation in command-driven workflows.

B. File and Language Handling

The implementation supports JavaScript extensions (.js, .mjs, .cjs, .jsx), Shell extensions (.sh, .bash, .zsh), Python (.py), PHP (.php, .php3, .php4, .php5), Java (.java), and Ruby (.rb). The scanner can skip excluded paths and binary files and applies configured maximum file-size and scan-depth restrictions.

C. Detection Engine

For each source file, the detection engine filters rules according to the detected language, compiles the applicable regular expressions, and executes matching with timeout protection. A match is converted into a structured finding containing rule information, severity, file path, line number, matched text, contextual snippet, confidence, and tags. File-level heuristics are then appended where applicable.

D. Security and Operational Safeguards

The implementation includes several defensive controls intended to prevent the scanner itself from being adversely affected by hostile or unusually large input. These controls include maximum file-size limits, regular-expression timeouts intended to reduce runaway matching and ReDoS exposure, maximum Base64 decoding size, maximum scan depth, path-traversal validation, and maximum findings per rule.

```
PS D:\root\github\malice-scan> python scan.py .\sample_project\
INFO: No output file specified, using default: D:\root\github\malice-scan\reports\sample_project_report.json
INFO: Initializing detector with patterns from: D:\root\github\malice-scan\patterns.json
INFO: Using configuration from: D:\root\github\malice-scan\config.json
INFO: Loading patterns from: D:\root\github\malice-scan\patterns.json
INFO: Loaded 40 patterns
INFO: Scanning: D:\root\github\malice-scan\sample_project
INFO: Starting scan of: D:\root\github\malice-scan\sample_project
INFO: Found 1 issues in: D:\root\github\malice-scan\sample_project\example_backdoor.js
INFO: Found 2 issues in: D:\root\github\malice-scan\sample_project\example_shell.sh
INFO: Found 2 issues in: D:\root\github\malice-scan\sample_project\example_sql_injection.js
INFO: Scan complete: 3 files scanned, 0 files skipped, 5 findings
INFO: Writing JSON report to: D:\root\github\malice-scan\reports\sample_project_report.json

Scan summary:
  high: 4
  low: 1
Total findings: 5

Full report written to: D:\root\github\malice-scan\reports\sample_project_report.json
PS D:\root\github\malice-scan>
```


```
{} sample_project_report.json > [ ] findings > { } 2
{
  "generated_at": "2026-06-22T13:34:11.2741692Z",
  "scanned_folder": "D:\\root\\github\\malice-scan\\sample_project",
  "finding_count": 5,
  "findings": [
    {
      "rule_id": "R004",
      "rule_name": "Suspicious eval-like usage",
      "description": "Use of eval, Function constructor, setTimeout with string, new Function, or similar",
      "severity": "high",
      "file": "D:\\root\\github\\malice-scan\\sample_project\\example_backdoor.js",
      "line": 11,
      "match": "eval(",
      "snippet": "  req.on('end', () => {\n    // backdoor: evaluate body content\n    eval(body);\n    res.end('ok');\n  });",
      "confidence": 0.98,
      "tags": [
        "eval",
        "code-exec"
      ]
    },
    {
      "rule_id": "R001",
      "rule_name": "Hardcoded credential - generic",
      "description": "Hardcoded username/password or token-like literal in assignment or export statements.",
      "severity": "high",
      "file": "D:\\root\\github\\malice-scan\\sample_project\\example_shell.sh",
      "line": 3,
      "match": "PASSWORD=\"hunter2\"",
      "snippet": "#!/bin/bash\n# example shell script with suspicious behavior\nPASSWORD=\"hunter2\"\ncurl -X POST http://malicious.exa
      "confidence": 0.98,
      "tags": [
        "hardcoded-credentials"
      ]
    },
    {
      "rule_id": "R008",
```


Figure 3. Illustrative report and review workflow.

VI. RESULTS AND DISCUSSION

A. *Functional Outcomes*

The implemented system demonstrates the complete workflow from source-project selection to multi-format finding reports. Functional capabilities for the project include recursive discovery of supported files, exclusions, language-aware rule selection, pattern matching, heuristic checks, confidence estimation, severity filtering, code-context snippets, console summaries, and JSON/HTML/CSV reporting. The HTML output is to provide severity-oriented filtering, search, tags, responsive presentation, and finding details, while the CSV output supports spreadsheet analysis and the JSON output supports structured consumption.

B. Testing Assets

Separate tests were developed for detector logic, report generation, and utility functions, supported by fixtures containing vulnerable and clean samples. A comprehensive test project was created with intentional issues across six languages: Java, JavaScript, PHP, Python, Ruby, and Shell. The distribution totals 77 intentional vulnerabilities. This provides implementation-level evidence that the project includes multi-language test material.

Table II. Comprehensive Test Project

Language / File Type	Intentional Issues
Java	10
JavaScript	10
PHP	17
Python	14
Ruby	21
Shell	5
Total	77

C. Performance Observations

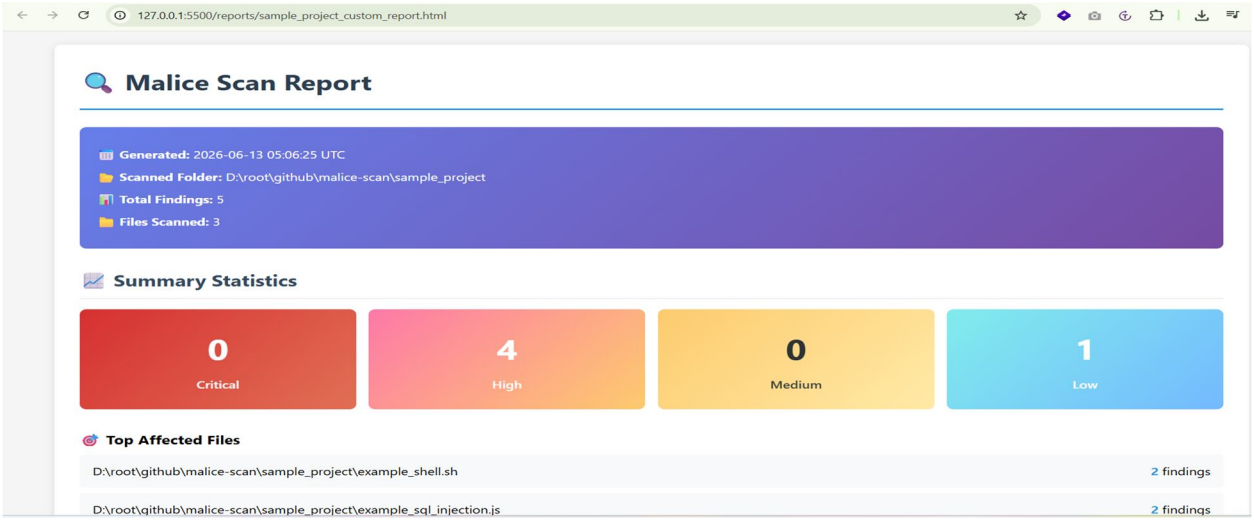
The project includes indicative benchmark figures for small, medium, and large projects: approximately 0.5 s and 50 MB for 10 files/1,000 LOC; approximately 5 s and 100 MB for 100 files/10,000 LOC; and approximately 50 s and 200 MB for 1,000 files/100,000 LOC.

Table III. Indicative Performance Figures

Project Scale	Approx. Scan Time	Approx. Memory
10 files / 1,000 LOC	0.5 s	50 MB
100 files / 10,000 LOC	5 s	100 MB
1,000 files / 100,000 LOC	50 s	200 MB

D. Discussion

The results support the practical role of Malice-Scan as a configurable pattern-oriented inspection tool. Its modular design allows rule definitions and configuration to be separated from core orchestration, while multiple report formats make findings accessible to different workflows. The test organization and intentionally vulnerable samples provide a basis for functional validation. Nevertheless, the available project does not establish a controlled comparison against other SAST tools or a ground-truth evaluation of all test cases. Consequently, the system should be viewed as a practical static scanner whose findings require security review and validation rather than as a complete vulnerability oracle.



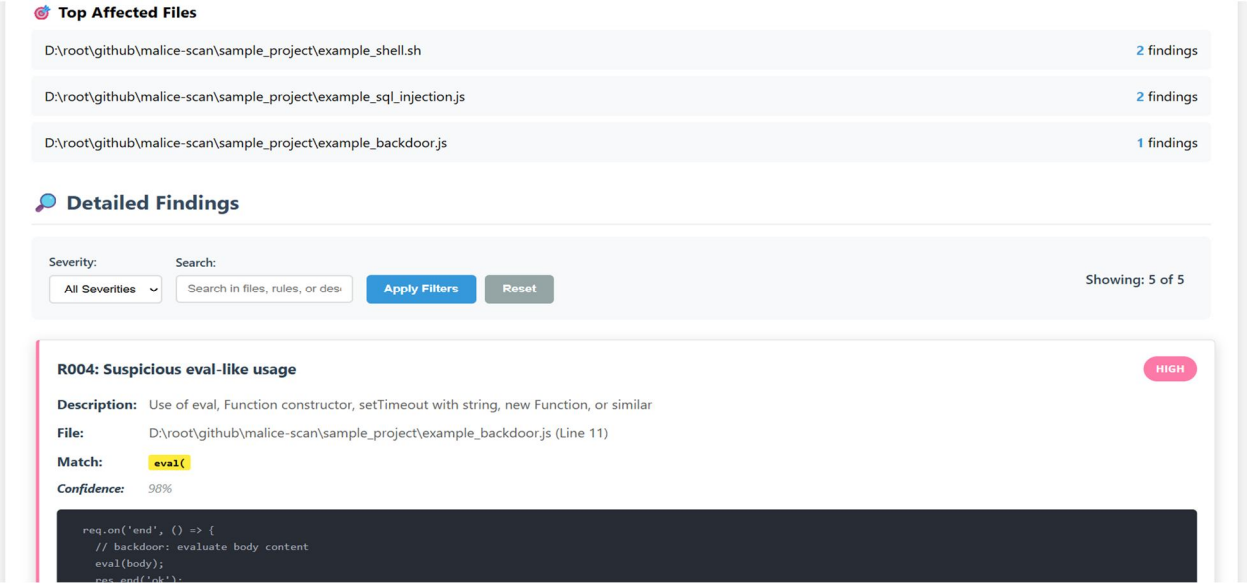


Figure 4. Example Malice-Scan HTML report or scan-result screen.

VII. ADVANTAGES AND LIMITATIONS

A. *Advantages*

The principal advantages are multi-language support; a clearly separated modular architecture; configurable JSON-based rules and settings; multiple reporting formats; severity filtering; contextual code snippets; selected heuristic checks; and operational safeguards for file size, scan depth, Base64 processing, and regular-expression execution. The command-line design also supports straightforward use in local review and CI/CD examples.

B. *Limitations*

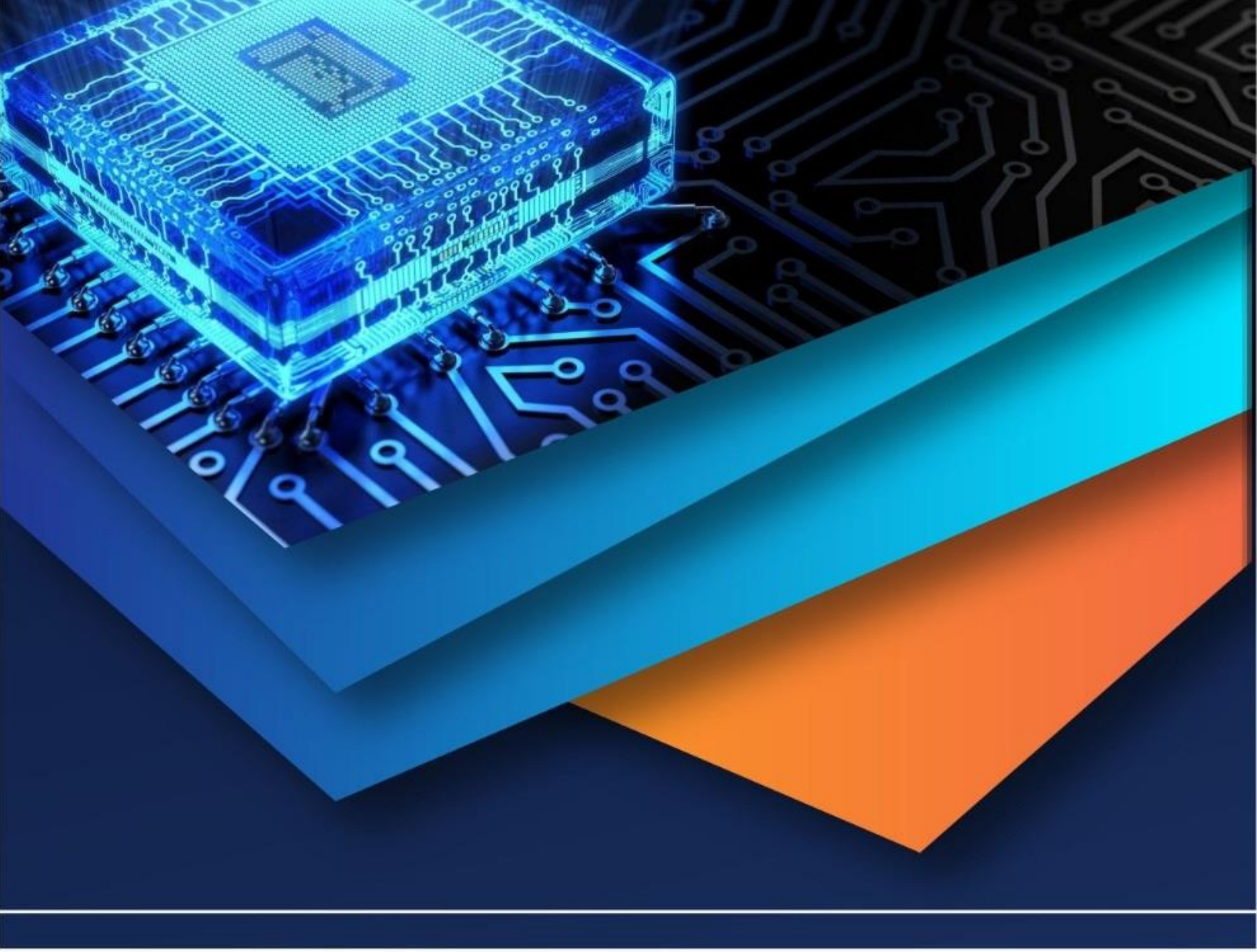
The scanner is primarily pattern based and therefore depends on the completeness and quality of its rule set. Regular-expression matching does not provide the same semantic, control-flow, or data-flow understanding as advanced static analyzers. The heuristics can indicate suspicious characteristics but cannot prove malicious intent or exploitability. False positives and false negatives remain possible, and business-logic and architecture-level weaknesses may not be identified.

VIII. CONCLUSION AND FUTURE SCOPE

This paper presented Malice-Scan, a configurable multi-language static security scanner based on forty regular-expression detection patterns, file-level heuristics, confidence estimation, configurable scanning controls, and JSON/HTML/CSV reporting. The implemented architecture separates orchestration, detection, configuration, utilities, and reporting, allowing the system to scan supported projects and present findings in forms suitable for security review and automation. The testing assets include intentionally vulnerable samples across all six supported languages, while the available evidence supports functional evaluation rather than claims of quantified detection accuracy. Future work should focus on strengthening the evidence base and extending analysis depth. Supported directions include systematic evaluation against labeled benchmarks, explicit measurement of false positives and false negatives, reproducible performance experiments, expanded and validated rule sets, improved language-aware parsing, data-flow and control-flow analysis, richer handling of encoded or obfuscated content, and tighter CI/CD feedback mechanisms. These enhancements should be evaluated empirically before claims of improved detection effectiveness are made.

REFERENCES

[1] B. Chess and G. McGraw, “Static Analysis for Security,” IEEE Security & Privacy, vol. 2, no. 6, pp. 76–79, 2004, doi: 10.1109/MSP.2004.111.
[2] OWASP Foundation, OWASP Code Review Guide, Version 2.0, methodology and source-code scanning guidance. Available: OWASP Project documentation.
[3] OWASP Foundation, OWASP Web Security Testing Guide, guidance on static source-code review and limitations of automated source-code analysis.
[4] G. Bennett, T. Hall, E. Winter, and S. Counsell, “Semgrep*: Improving the Limited Performance of Static Application Security Testing (SAST) Tools,” in Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE ’24), 2024, pp. 614–623, doi: 10.1145/3661167.3661262.
[5] “Detecting security vulnerabilities with static analysis – A case study,” Acta Polytechnica Hungarica / related published case-study record, 2021, doi: 10.1556/606.2021.00454.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY