IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 7, October 2026
Impact Factor: 8.2

Chakraview: A Hybrid Static-Analysis and Local-Llm Prototype for Vulnerability Detection and Mitigation Guidance in Open-Source Software

Amit Rushikesh Tayade[1], Ajay Vinod Dawkhare[2], Aryan Mandar Wable[3], Mayur Shashikant Thakare[4], Sakshi Sopan Bhor[5]
Students, Department of AI and Data Science[1-4]
Guide, Department of AI and Data Science[5]
Jaihind College of Engineering, Kuran, India
[1] tayadeamit16@gmail.com, [2] ajayvdawkhare2004@gmail.com
[3] aryanwable67@gmail.com, [4] thakarem154@gmail.com
[5] sakshibhor2604@gmail.com

Abstract: *Open-source software benefits from rapid, distributed development, but reviewing it for security weaknesses is labour-intensive, and developers need findings that are explained and actionable. We present Chakraview, a prototype that combines a rule-based static stream with a locally hosted large language model (DeepSeek-Coder 6.7B Instruct served through Ollama). Uploaded Python, C/C++, PHP, and Java files or ZIP archives are parsed with Tree-sitter, analysed by both streams, fused by a weighted score (0.6 × LLM confidence + 0.4 × static flag), and reported with CWE labels, explanations, and suggested mitigations. We document the architecture, analyse the fusion rule, and report two preliminary checks: seven functional checks recorded in project documentation, and a reproducible measurement of the static stream on four developer-written files containing 29 seeded vulnerable lines, of which 23 were flagged (22 under the correct CWE); all misses were injection-type flaws needing data-flow reasoning. The LLM stream, the fusion score, and the generated fixes have not been evaluated on labeled data, and suggested fixes are not compiled, tested, or rescanned. Chakraview is therefore a preliminary system prototype, not a validated detector. We specify a held-out evaluation protocol comparing static-only, LLM-only, and fused analysis, with separate tests of localization, calibration, and mitigation correctness.*

Keywords: software security, vulnerability detection, static analysis, large language models, vulnerability mitigation, open-source software

I. INTRODUCTION

Open-source software underpins much of modern infrastructure, and its rapid, distributed development means security review must be repeated continuously. A useful review tool must do more than flag suspicious syntax: a finding is valuable when it points to the affected code, explains the risk in context, and gives the developer a safe next step. Rule-based static analysis supplies repeatable checks and is long-established security practice [1], but developers report that false positives and poorly presented warnings limit its adoption [2]. Large language models (LLMs) can explain code and propose changes, which addresses the presentation problem, but their verdicts and repairs need independent verification.

Prior evaluations caution against treating LLM output as a dependable security verdict. In a multitask evaluation on Big-Vul, existing state-of-the-art and pretrained language-model approaches generally outperformed LLMs on detection, although some LLMs did better on assessment or localization when supplied with additional context [3]. A

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 7, October 2026 Impact Factor: 8.2

study of PHP vulnerabilities reported that even its best-performing tested model had a substantial false-positive rate, alongside missed and partially detected cases [4]. At repository scale, JitVul links functions to vulnerability-introducing and fixing commits; its findings show that separating vulnerable from benign code remains difficult and that models can over-analyze security guards [5]. Work on more realistic, de-duplicated benchmarks further reports that code language models perform markedly worse than earlier datasets suggested [6].

The same caution applies to mitigation. Repairing a vulnerability is not equivalent to generating plausible code. An evaluation of LLM-based repair on real-world C/C++ examples found that performance depended on repair complexity, with harder cases requiring reasoning about broader program semantics [7]. A study of Java vulnerability repair reported that many generated patches were not compilable and that the evaluated systems fixed only a minority of benchmark vulnerabilities [8]. LLM assistants can also emit insecure code themselves [9]. These results motivate a design in which the model proposes a mitigation while program analysis, tests, and developer review remain responsible for verification.

This paper presents **Chakraview**, a prototype that combines a rule-based static stream with local LLM analysis to inspect uploaded source code and produce vulnerability reports with mitigation guidance. We are deliberately conservative about what the evidence supports, and organise the paper around three questions. **RQ1:** how are parser context, rules, and a local LLM combined, and what does the fusion rule imply for ranking and corroboration? **RQ2:** how much of a set of known, seeded vulnerabilities does the rule stream alone recover, and where does it fail? **RQ3:** what evidence is needed before the LLM stream, the fusion score, and the generated fixes can be called effective? The contributions are:

- A description of the architecture and fusion rule, derived from the source code, including behaviours that differ from earlier project descriptions (Section IV).
- An analysis of the fusion rule showing its score ranges, thresholds, corroboration condition, and failure modes (Section IV-E).
- A reproducible measurement of the static stream on 29 seeded vulnerable lines in four sample files, with a miss analysis (Section VII-B).
- A concrete evaluation protocol for detection, localization, calibration, and mitigation correctness (Section IX-B).

No claim of benchmark-level accuracy or verified automated repair is made. Section II reviews related work, Section III states the problem and objectives, Section IV describes the architecture, Section V the methodology, Section VI the experimental setup, Section VII the results, Section VIII applications and advantages, Section IX limitations and future scope, and Section X concludes.

II. RELATED WORK

A. Static Analysis for Security

Static analysis inspects code without executing it and has long been recommended as part of secure development [1]. Pattern-based tools such as Semgrep express rules as code patterns and are easy to extend [10]. Their weaknesses are well documented: warnings that are numerous, imprecise, or hard to interpret reduce developer trust [2]. Weakness taxonomies such as CWE [11] and the OWASP Top 10 [12] give findings a shared vocabulary, which Chakraview adopts for its labels.

B. Learning-Based and LLM-Based Detection

Deep-learning detectors learn from code representations: VulDeePecker uses code gadgets [13], and Devign applies graph neural networks to program graphs [14]. Datasets such as Big-Vul [15] and CVEfixes [16] link vulnerable code to CVE records and fixing commits, which makes commit-level splits possible. LLM performance varies by task, context, language, and test set. The Big-Vul multitask evaluation found that LLMs were not uniformly stronger than specialised or pretrained alternatives [3]; the PHP study stresses including benign or patched examples when measuring

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 7, October 2026
Impact Factor: 8.2

detection [4]; and JitVul adds repository-level interactions and commit-linked pairs [5]. Chakraview uses the open DeepSeek-Coder instruction model [17] because it can run locally. Its parser-derived context should not be mistaken for full interprocedural or repository-wide data-flow analysis, so it must be evaluated at the code-unit scope it actually supports.

C. Vulnerability Repair and Mitigation

Remediation has stricter success conditions than explanation: a patch should remove the vulnerability while preserving intended behaviour [18]. Zero-shot LLM repair has been explored on synthetic and real examples [19], and fine-tuned models such as VulRepair target vulnerability-specific patches [20]. In real-world C/C++ experiments, LLMs were stronger on simple, localised repairs and weaker on complex cases [7]. Hybrid systems follow a detect–repair–validate workflow in which static analysis checks a model-generated candidate rather than treating generation as proof [21]. Chakraview currently generates mitigation guidance and does not implement the validation step; the distinction between advice and verified repair defines its scope.

D. Confidence and Calibration

A rule that adds a model confidence to a static flag assumes the confidence is meaningful. Modern neural networks can be poorly calibrated [22], and confidence verbalised by an LLM tends toward overconfidence [23]. Chakraview's LLM confidence is a number the model writes in its answer, so calibration must be tested rather than assumed. Table I summarises the evidence used to shape the design and the evaluation protocol.

TABLE I: Evidence from prior work and its design implication

Ref.	Reported finding	Implication for Chakraview
[3]	LLMs not uniformly better than specialised models; task-dependent	Score detection and localization separately
[4]	Substantial false positives and misses on PHP	Include benign and patched code
[5]	Vulnerable vs. benign hard to separate at repository scale	Use commit-linked pairs; state scope
[6]	Realistic, de-duplicated data lowers measured performance	Deduplicate; split by project or time
[7]	Repair success falls as complexity rises	Stratify mitigation results
[8]	Many patches non-compilable; minority fixed	Compile and test before accepting
[21]	Detect–repair–validate with static checking	Rescan candidates; keep human approval
[23]	Verbalised confidence tends to overconfidence	Calibrate C_{LLM} before use as probability

III. PROBLEM STATEMENT AND OBJECTIVES

This work addresses two problems. The first is review effort and explanation: security review of open-source code is repeated continuously, and rule-based tools return terse findings that developers must interpret, whereas language models explain well but are not reliable verdict-givers. The second is an evidence gap: tools that combine rules with a language model are usually described by their features, not by validated measurements of what each part contributes.

The primary objectives of Chakraview are (i) to combine a rule-based stream and a local language-model stream in one pipeline that reports CWE-labelled findings with explanations and mitigation guidance; (ii) to keep model inference local, so uploaded source need not be sent to an external service; and (iii) to record, for every finding, which stream produced it, so that a reviewer can see the evidence behind a ranking. The secondary objectives of this paper are to distinguish clearly between demonstrated behaviour and configured or intended behaviour, and to specify the validation needed before stronger claims can be made.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 7, October 2026
Impact Factor: 8.2

IV. SYSTEM ARCHITECTURE

A. Layered Overview and Design Rationale

Four goals shaped the design. **Locality:** analysis of uploaded source should not require sending code to an external service, which is why the model runs under a local Ollama endpoint. **Complementarity:** rules are repeatable and cheap but blind to context, whereas an LLM can read a whole function but is stochastic; the two are combined so that each can corroborate the other. **Graceful degradation:** if the model service is unreachable or the model is not pulled, the LLM stream returns an empty result and the pipeline continues with static findings only. **Traceability:** every finding carries a tag recording which stream produced it, so a reviewer can see the evidence rather than a single opaque score. The design assumes a developer or reviewer remains in the loop: Chakraview prioritises and explains potential weaknesses; it does not decide that code is safe, and it does not change code.

Fig. 1 shows the pipeline. A Flask web interface accepts a source file or ZIP archive with a software name and version. Archives are extracted, and every file with a supported extension (.c, .cpp, .cc, .cxx, .h, .hpp, .py, .php, .java) found under the upload directory is analysed independently. Each file passes through two streams whose outputs are fused, ranked, stored with the scan metadata, and shown in a dashboard and as an Excel or JSON export.

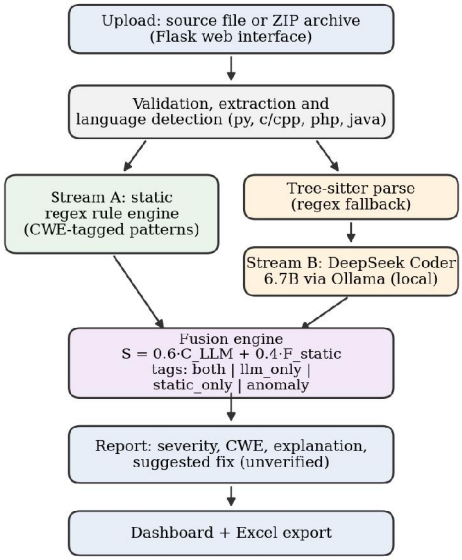


Fig. 1. Chakraview pipeline. Stream A (rules) and Stream B (Tree-sitter context plus a local LLM) run on each file; the fusion engine tags and ranks findings.

B. Context Extraction

Tree-sitter [24] parses each file, and the tree is converted into control-flow-style path strings such as param(user_id) → if_stmt → call_expr(execute). Node types of interest are function definitions, conditionals, loops, calls, argument lists, parameters, assignments, binary expressions, and returns. Up to ten paths are placed in the LLM prompt. When the Tree-sitter package is unavailable, a line-level regular-expression fallback is used. The paths are structural hints; each file is analysed in isolation, so cross-file and interprocedural taint tracking is not provided.

C. Stream A: Rule-Based Static Analysis

Stream A is a built-in regular-expression engine. A table of case-insensitive, line-level patterns, each carrying a CWE identifier, severity, and description, is applied to every line; language-specific routines for Python, Java, C/C++, and PHP add further checks. Covered classes include SQL injection (CWE-89), cross-site scripting (CWE-79), command injection (CWE-78), insecure deserialization (CWE-502), unsafe C string functions (CWE-120), hard-coded credentials (CWE-798), weak cryptography (CWE-327), and XXE and SSRF in Java. Semgrep appears in the dependency file and

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 7, October 2026
Impact Factor: 8.2

in design comments, but the submitted source does not invoke it; earlier descriptions of a Semgrep-based stream are therefore not supported by the code, and Semgrep integration is listed as future work. Table II inventories the rules.

TABLE II
Rule inventory of Stream A

Scope	Rules	CWE identifiers
Generic (all languages)	11 patterns, 8 classes	89, 79, 78, 502, 120, 22, 798, 327
Python	2 checks	Dangerous imports (502, 78, 95); eval/exec (95)
Java	2 checks	XXE (611); SSRF (918)
C / C++	2 checks	Format string (134); integer overflow (190)
PHP	2 checks	File inclusion (98); weak randomness (330)

Totals: 11 generic patterns and 8 language-specific checks covering 15 distinct CWE identifiers. Each rule operates on a single line; the PHP file-inclusion rule requires a request variable inside the include call itself.

D. Stream B: Local LLM Analysis

Stream B sends the file, its language, and the extracted paths to DeepSeek-Coder 6.7B Instruct [17], served locally by Ollama [25]. The prompt demands five fields: a VULNERABLE/CLEAN verdict, a CWE identifier, a short explanation, a complete corrected version of the code, and a confidence between 0 and 1. Inference uses temperature 0.1, a 4,096-token context, at most 1,024 generated tokens, and a 120 s timeout; code beyond 4,000 characters is truncated. The model therefore returns **one verdict and at most one CWE per file**.

Response parsing is lexical: the verdict is VULNERABLE if that word appears anywhere in the reply, the CWE is the first "CWE-*n*" string found, and confidence defaults to 0.5 when no number can be parsed. This tolerates format drift but can mis-set a verdict if the model writes the word inside a negated sentence. A VULNERABLE verdict with no parseable CWE is marked as an anomaly.

E. Signal Fusion

For each static finding the fusion score is

$$S = 0.6 \times C_{\text{LLM}} + 0.4 \times F_{\text{static}} \qquad (1)$$

where C_{LLM} is the LLM confidence if its verdict is VULNERABLE and its CWE string equals the finding's CWE (otherwise 0), and F_{static} is 1 for every static finding. Findings are tagged **both** (corroborated), **static_only**, **llm_only**, or **anomaly**, and sorted by severity first and S second. Table III gives the resulting cases and Fig. 2 plots the score as a function of confidence.

TABLE III: Fusion cases implied by the implementation

Tag	Score S	Condition and treatment
both	$0.4 + 0.6c$ (0.70–1.00)	Static finding; LLM VULNERABLE with identical CWE
static_only	0.40	Static finding; LLM CLEAN, failed, or different CWE
llm_only	$0.6c$; kept if ≥ 0.35 ($c \geq 0.583$)	LLM CWE unmatched; severity fixed to High, line 0
anomaly	$0.6c$; no threshold	VULNERABLE, no CWE, and the file has no static finding; severity Medium

c = LLM-reported confidence in [0, 1]; for *both*, S ranges from 0.40 (c = 0) to 1.00 (c = 1), and 0.70 at the default c = 0.5.

Five properties follow. First, S is a ranking score, not a probability: no calibration or benchmark-derived justification for the 0.6/0.4 weights or the 0.35 threshold is documented. Second, corroboration is exact string equality of CWE labels, so related labels (for example CWE-120 vs. CWE-121 or CWE-787) do not corroborate; the same line can then appear twice, once as static_only and once as llm_only. Third, because sorting is by severity first, S only orders findings within a severity band, and llm_only entries are always High regardless of CWE. Fourth, a static-only hit (0.40) outranks any LLM-only finding with $c < 0.667$, so the LLM cannot demote a static false positive; it can only add

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 7, October 2026
Impact Factor: 8.2

or corroborate. Fifth, a VULNERABLE verdict with no parseable CWE corroborates *every* static finding in the file: each is tagged both and scored 0.4 + 0.6*c*, and the separate anomaly entry is produced only when the file has no static finding. A single vague model answer can therefore raise the score of all rule hits at once.

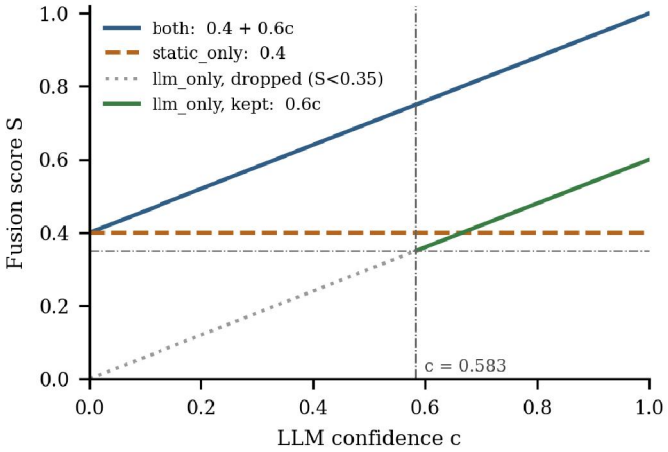


Fig. 2. Fusion score *S* versus LLM confidence *c* for each tag. The dotted segment is dropped by the 0.35 threshold ($c < 0.583$).

F. Mitigation Guidance and Reporting

For corroborated findings, the report attaches the model's corrected code and explanation. For static_only findings, if the model is available, a second targeted request supplies the CWE and line and asks for a fix. A rule-based remediation library additionally provides guidance text and example code per vulnerability class. Each fused finding is a record holding the weakness type, CWE, severity, line number, code snippet, description, suggested fixed code, detection tag, fusion score, LLM confidence, and static flag, so every report row can be traced to the stream that produced it. Reports contain Vulnerabilities, Summary, and Remediation sheets. This is mitigation guidance, not a validated patch: the code does not compile, test, or rescan any suggested change, and the output should be treated as a candidate for developer review.

G. Cost Model

The pipeline issues one LLM request per file, plus one additional targeted-fix request for every static_only finding when the model is available. A file with *k* static_only findings can therefore need up to 1 + *k* requests, each bounded by the 120 s timeout, so worst-case latency grows with the number of rule hits rather than with file size alone. On a CPU-only machine, local 6.7B-parameter inference can be slow enough that large archives become impractical. We have not measured runtime, memory, or throughput, and the protocol in Section IX-B requires them to be reported.

H. Deployment Boundary and Documentation Corrections

Inference runs against a local Ollama endpoint, and no code path in the reviewed source calls an external LLM service, although the configuration file lists an external-API option. Two corrections to earlier project descriptions are needed. First, the source does not query the NVD [26]: the CVE field is filled from a fixed lookup keyed on vulnerability class that returns the first listed identifier, and the NVD-details method returns placeholder values. The CVE column must therefore not be read as evidence that a given CVE applies to the analysed code; CWE is the meaningful label. Second, the static stream is regex-based (Section IV-C).

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 7, October 2026
Impact Factor: 8.2

V. METHODOLOGY

A. Request Workflow

An analysis passes through the six stages of Fig. 3. The user uploads a source file or ZIP archive; the system validates the upload, extracts any archive, and detects the language; the rule stream and the Tree-sitter-plus-LLM stream analyse each file independently; the fusion engine tags and scores the findings; explanations and suggested fixes are attached; and the report is shown in the dashboard or exported. Because each stage can fail independently (a failed parse, an unreachable model, an unparseable reply), the evaluation in Sections VII-B and VII-C examines the rule stream and the fusion logic separately and does not report a single success rate.

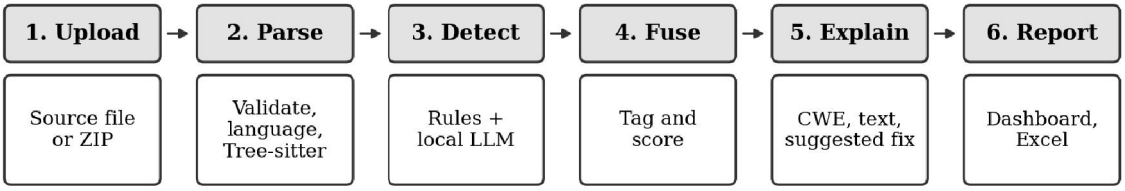


Fig. 3. Request workflow of Chakraview.

B. Categories of Evidence

Four categories of evidence are kept apart, because they exercise different code paths: (i) seven functional checks recorded in the project documentation; (ii) a measurement of the rule stream alone on seeded sample files; (iii) a test of the fusion function with injected model outputs; and (iv) a proposed held-out evaluation, which has not been run. Categories (i) to (iii) concern components; none estimates the accuracy of the language model, so none supports a claim of detection performance for the whole system.

C. Static-Stream Measurement Design

To obtain at least one measurement that a reader can reproduce, we ran Stream A alone, with no LLM and no network, on the four sample files in the repository (Python, C, PHP, Java; 152 lines in total). Each file was written by the project team with comments marking vulnerabilities, so we treat the marked lines as ground truth: 29 seeded vulnerable lines. A seeded line counts as *flagged* if any finding is reported on it, and as *correct CWE* if one of those findings carries the expected CWE. The script (static_stream_check.py) identifies seeded lines by code text, not line number, and is provided with the paper. Results are given in Section VII-B.

D. Fusion Test Design

The language model was not available in the measurement environment. The fusion function was therefore exercised directly with injected model outputs (a verdict, a CWE, and a confidence) on the sample files. This tests the fusion logic deterministically and says nothing about what the model would return.

VI. EXPERIMENTAL SETUP

The evaluation was limited to local software checks: no user study, no load test, and no live-model accuracy test was performed. The rule-stream measurement and the fusion test ran in a Linux container (Ubuntu 24.04, Python 3.12, one CPU, 3 GB memory) with no network, no Ollama service, and the regular-expression fallback in place of Tree-sitter, because the Tree-sitter language package was not installed.

Table IV lists the settings that must be frozen and reported for any evaluation run. They are taken from the submitted source and configuration, and are the minimum needed to repeat a run.

TABLE IV : Implementation summary

Component	Setting in submitted code
Runtime / UI	Python; Flask dashboard, upload, scan-detail views
Input	.c .cpp .cc .cxx .h .hpp .py .php .java .zip
Parsing	Tree-sitter (regex fallback); ≤ 10 paths per prompt

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 7, October 2026
Impact Factor: 8.2

Component	Setting in submitted code
Stream A	Regex pattern table + per-language routines; CWE, severity per rule
Stream B model	deepseek-coder:6.7b-instruct via Ollama (localhost)
Inference	Temp. 0.1; context 4,096; ≤ 1,024 new tokens; 120 s timeout
Prompt limit	4,000 characters per file, then truncated
Fusion	Weights 0.6 / 0.4; llm_only threshold 0.35
Output	Dashboard; Excel (3 sheets); JSON
Licence	MIT
Measurement host	Ubuntu 24.04; Python 3.12; 1 CPU; 3 GB; no Ollama

Two reproducibility gaps should be closed before a conference artefact is released: the dependency file lists packages the Ollama path does not use (for example PyTorch and Transformers) while omitting the Flask and Tree-sitter-language packages the application imports, and the supported-language list in the configuration (six languages in Stage I) is wider than the four languages the upload route and static routines handle.

VII. RESULTS AND DISCUSSION

A. Reported Functional Checks

The project documentation records seven functional checks as passing (Table V). They indicate that the listed components execute on prepared examples. They do not estimate detection performance: no test-set size, benign-example count, false-positive or false-negative count, precision, recall, F1, runtime, or baseline comparison is available. The documentation is not part of the submitted archive, and the static rules in the submitted code label unsafe string copies as CWE-120 rather than CWE-121, so the CWE-121 outcome presumably comes from the LLM stream or a different build and should be reconciled before submission.

TABLE V: Functional checks recorded in project documentation

Check	Input / component	Reported outcome
SQL injection	Python file	CWE-89 and location reported
Buffer overflow	C file	CWE-121, critical severity
Command injection	PHP file	CWE-78 and snippet reported
LLM assessment	Java example	Vulnerable verdict returned
Fusion behaviour	LLM-only finding	Finding retained
Report generation	Excel export	Vulnerability and remediation sheets
Web interface	Results dashboard	Severity summary displayed

B. Static-Stream Measurement on Seeded Samples

Table VI and Fig. 4 give the results of the rule-stream measurement by file, and Table VII gives them by weakness class. The measurement design is described in Section V-C.

TABLE VI: Static-stream results on the four sample files

File	Seeded	Flagged	Correct CWE	Unseeded flags
Python	9	7	7	2 (import lines)
C	6	6	5	0
PHP	8	6	6	0
Java	6	4	4	1 (method signature)
Total	**29**	**23 (79.3%)**	**22 (75.9%)**	**3**

Of 26 distinct flagged lines, 23 were seeded (88.5%). This is not a false-positive rate: the samples contain no benign code.

Copyright to IJARSCT
www.ijarsct.co.in
DOI: 10.48175/IJARSCT-40460

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 7, October 2026
Impact Factor: 8.2

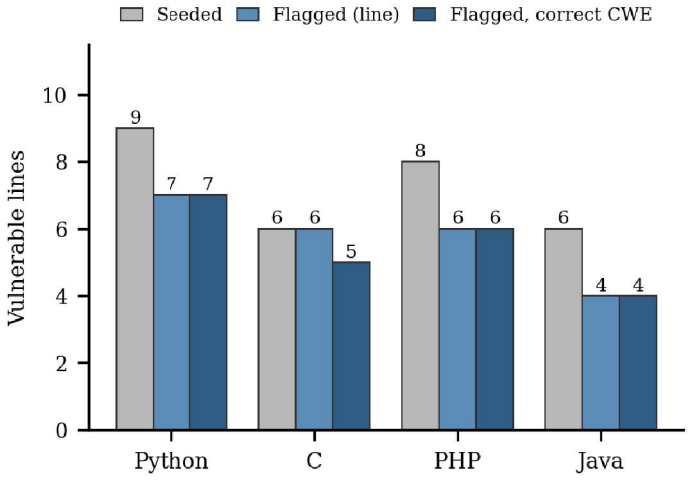


Fig. 4. Seeded vs. flagged vulnerable lines per sample file, with and without the correct CWE (Stream A only).

Where the rules succeed. The 23 flagged lines are dominated by constructs identifiable from a single token or call: unsafe C functions (strcpy, strcat, gets), eval and exec, pickle.loads and unserialize, hard-coded keys, weak hashes and random generators, an unsafe XML parser factory, and URL construction from a parameter.

Where they fail. All six misses are injection-type flaws in which untrusted data reaches a sensitive call through concatenation or interpolation: SQL concatenation in Python, PHP, and Java; an f-string passed to os.system; a concatenated Runtime.exec call; and a PHP include whose path passes through an intermediate variable. The SQL and command patterns assume a "$" inside the same line, and the include rule needs a request variable inside the call itself, so none of them follows data through concatenation, interpolation, or a variable assignment. One further error is a wrong root cause: the use-after-free line is flagged only as CWE-120 because it contains strcpy. The three unseeded flags come from the Dangerous Import rule firing on two import statements and from the SSRF check matching a method signature; whether such flags are false positives or acceptable advisories is a policy choice to settle in the protocol.

TABLE VII: Stream A recall by weakness class (correct CWE)

Weakness class	Seeded	Recovered
SQL injection (CWE-89)	3	0
OS command injection (CWE-78)	4	2
File inclusion (CWE-98)	1	0
Use after free (CWE-416)	1	0 (flagged as CWE-120)
eval / exec code injection (CWE-95)	2	2
Cross-site scripting (CWE-79)	1	1
Hard-coded credentials (CWE-798)	6	6
Insecure deserialization (CWE-502)	2	2
Weak crypto / randomness (CWE-327, 330)	2	2
C buffer, format, integer (CWE-120, 134, 190)	5	5
XXE and SSRF (CWE-611, 918)	2	2
Total	**29**	**22**

Copyright to IJARSCT
www.ijarsct.co.in
DOI: 10.48175/IJARSCT-40460

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 7, October 2026
Impact Factor: 8.2

C. Fusion Behaviour on Injected Model Outputs

The LLM was not available in our environment, so we exercised the fusion function directly with injected model outputs, which tests the logic but says nothing about what the model would actually return. Table VIII lists six cases run through the submitted fusion code on the sample files. They confirm the analysis of Section IV-E and expose a localization gap: an llm_only entry carries line number 0 and a 120-character snippet from the start of the file, so a vulnerability found only by the model cannot yet be pointed to in the code.

TABLE VIII: Fusion code run with injected LLM outputs

Sample and case	Injected output	Resulting entries
C, strcpy line	CWE-120, $c = 0.9$	All four CWE-120 lines tagged both, S = 0.94 (including the use-after-free line)
C, use-after-free line	CWE-416, $c = 0.8$	static_only CWE-120 (0.40) plus llm_only CWE-416, High, line 0 (0.48): two entries, correct class at no line
Python, SQL concatenation (static miss)	CWE-89, $c = 0.8$	llm_only CWE-89, line 0, S = 0.48
Same case, low confidence	CWE-89, $c = 0.5$	No entry: S = 0.30 < 0.35, dropped silently
Python, vague verdict	VULNERABLE, no CWE, $c = 0.5$	All 9 static findings tagged both, S = 0.70; no anomaly entry
File with no static finding	VULNERABLE, no CWE, $c = 0.5$	One anomaly entry, Medium, S = 0.30

D. Interpretation and Hypotheses

These numbers describe a rule set on 152 lines written by its own authors, with no benign code and no held-out data; they show where the rules are weak, not how the system performs on real projects. They do suggest a testable role for the LLM stream: the misses are the data-flow-dependent injection cases, where a model reading the whole function might help. We state this as hypotheses for the protocol in Section IX-B, not as results: **H1**, fusion recovers more injection-type vulnerabilities than static-only; **H2**, fusion does not increase false positives on patched and benign code beyond a stated tolerance; **H3**, LLM confidence separates true from false findings better than chance after calibration.

VIII. APPLICATIONS AND ADVANTAGES

Practical value. Chakraview's present value is as an integrated workflow that places a finding, its CWE label, an explanation, and a candidate fix in one report, with local inference so that source code need not leave the machine in the default configuration. The seeded-sample analysis shows that the rule stream alone is easily satisfied by token-level patterns and weak on injection flaws, which is where the semantic stream is expected to contribute. Whether it does is unmeasured.

Intended workflow. A maintainer uploads a repository snapshot, reads the dashboard's severity summary, and opens the highest-ranked findings. For a both-tagged finding the CWE, the rule hit, and the model's explanation agree and the item is a good starting point; a static_only finding should be checked against the source line; an llm_only or anomaly finding is a lead that must be located by hand. Suggested fixes are compared with the code, tested by the maintainer's own test suite, and rescanned before any commit.

Positioning. Chakraview sits between two lines of work. Compared with rule-only scanners [10], it adds a model that can read a whole function and explain a finding in plain language. Compared with detect–repair–validate systems [21], it implements the first two steps but not the third, which is the step that gives such systems their evidential value. Compared with learned detectors trained on labeled corpora [14], it uses a general code model prompted at inference time and no task-specific training, which lowers the barrier to adoption but removes the benchmark-derived evidence

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 7, October 2026
Impact Factor: 8.2

those detectors can report. The honest summary is that Chakraview's design is plausible and its integration is working, while its effectiveness is an open empirical question.

IX. LIMITATIONS AND FUTURE SCOPE

A. Limitations

Over-interpretation risks. The principal risk is reading the fusion score as a probability. A static flag and a verbalised confidence are not commensurate, and a weighted sum is not calibrated. Reports should show each finding's source tag and keep the underlying evidence rather than a single risk number. The mitigation component has a similar boundary: an untested repair may be incomplete, break behaviour, or leave the root cause in place. The proposed generate–check–rescan–approve loop, with precedent in hybrid repair research [21], should be built and evaluated before Chakraview is described as a repair tool.

Threats to validity. *Construct:* "flagged" means a finding on a seeded line, which is weaker than a correct, actionable report. *Internal:* the samples and rules share authors, so rule coverage may be tuned to the samples; the LLM stream was not run in our measurement. *External:* 152 lines in four languages say little about large open-source repositories, and one model (6.7B parameters) was used. *Reproducibility:* sampled LLM output, Ollama and model versions, and the dependency-file gaps in Section VI can change results.

Security of the tool itself. Because uploaded code is inserted into the model prompt, the code can address the model: a comment instructing it to report the file as CLEAN is a prompt-injection attempt, and the lexical response parser would accept whatever verdict text results. The rule stream is immune to this but also blind to anything that evades its patterns, which is a further reason to keep both streams and to report disagreement rather than resolve it silently. Archives are extracted on the server, so extraction limits, size caps, and authentication belong in any deployment beyond a trusted local machine. The pipeline reads uploaded code and never executes it, and dynamic analysis is disabled in the configuration.

Ethics and responsible use. The tool processes uploaded source code, which may be proprietary or contain secrets; local inference reduces exposure, but stored scans, uploads, and exports should be access-controlled and deletable. Vulnerability reports are dual-use, so findings in third-party projects should be disclosed to maintainers responsibly. Suggested fixes should never be applied automatically, and the placeholder CVE field should be disabled or removed to avoid misleading users.

B. Proposed Evaluation Protocol

The following protocol is proposed future work; it is not an experiment completed for this paper. Table IX maps each research question to the data, comparison, and measures the protocol requires.

TABLE IX: Evaluation plan by research question

Question	Data and comparison	Measures and tests
RQ1 (fusion)	Static-only vs. LLM-only vs. fused, identical held-out cases	Precision, recall, F1, FPR on benign code; bootstrap CIs; McNemar; ablation of weights and threshold
RQ2 (rules)	Rule stream on commit-linked vulnerable, patched, and benign code	Recall by CWE and language; localization hit rate; flags per KLOC
RQ3 (LLM, fixes)	LLM stream and its fixes on the same cases	Calibration (reliability, ECE); compile and test pass rate; rescan clean rate; rater agreement

The following steps implement this plan.

- **Corpus and scope.** Use a named, versioned dataset such as Big-Vul [15], CVEfixes [16], PrimeVul [6], or the Juliet suite [27]. State the unit of analysis (function or file), and include vulnerable, benign, and patched versions, preserving the commit link where available. Record language, CWE, project, and commit.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 7, October 2026
Impact Factor: 8.2

- **Leakage control.** Keep prompt-development examples apart from test examples; split by project or fixing commit, deduplicate near-identical code, and report any filtering.
- **Frozen configuration.** Record model tag, prompt, parser version, rule set, weights, threshold, hardware, and inference settings. Repeat runs to measure output variance, since the model is sampled at temperature 0.1.
- **Component comparison.** Evaluate static-only, LLM-only, and fused configurations on identical held-out cases. Report true positives, false positives, false negatives, precision, recall, F1, and false-positive rate on benign code, stratified by language and CWE, with bootstrap confidence intervals and McNemar tests for paired comparisons. Choose weights and thresholds on a validation split only.
- **Calibration.** Plot reliability diagrams and report expected calibration error for C_{LLM} [22], and compare against a calibrated or rank-based alternative to Eq. (1).
- **Localization and mitigation.** Score whether the labeled vulnerable region is identified, not only the class. For each suggested fix, record compilation or syntax validity, regression-test outcome, a security test or proof-of-vulnerability check, and a static rescan; two reviewers should rate semantic appropriateness, with agreement reported. Report partial, failed, and harmful repairs separately.
- **Adversarial inputs.** Include files with embedded instructions in comments and with obfuscated sinks, and report how often the verdict changes.
- **Artefacts.** Release the manifest, prompts, rule configuration, raw outputs, rubric, and scripts, and document how weakness labels map to CWE or CVE records.

C. Roadmap

Future work follows the evidence gaps and is grouped by priority in Table X.

TABLE X: Roadmap grouped by priority

Priority	Actions
Mandatory corrections	Remove or implement the NVD/CVE step; reconcile the supported-language list and dependency file; describe Stream A as regex-based or integrate Semgrep; run the LLM stream and the held-out evaluation before claiming accuracy.
Possible modifications	Add intra-file taint rules for injection patterns; match CWE families rather than exact strings; calibrate or replace the 0.6/0.4 weights; record line numbers for llm_only findings; add the compile–test–rescan validation loop.
Future extension	Code property graphs [28] for interprocedural reasoning; repository-level indexing; the Stage II languages in the configuration; fine-tuning or larger models; a user study of reviewer effort.

X. CONCLUSION

This paper described Chakraview, a prototype that combines rule-based static checks with local LLM analysis to identify potential vulnerabilities and generate mitigation guidance, and analysed its fusion rule. A reproducible static-stream measurement recovered 23 of 29 seeded vulnerable lines (22 with the correct CWE), with all misses in injection-type flaws, while the LLM stream, the fusion score, and the generated fixes remain unevaluated on labeled data. Chakraview should therefore be presented as a preliminary system prototype, not as a proven detector or repair tool.

Acknowledgment

The authors thank their project supervisor, the Department of Artificial Intelligence and Data Science, and Jaihind College of Engineering, Kuran, for the guidance, facilities, and academic environment that supported this work.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 7, October 2026
Impact Factor: 8.2

Declarations

Funding. The authors received no specific funding for this work. **Conflicts of interest.** The authors declare no conflicts of interest. **Ethics.** The study involves no human participants and no personal or proprietary data; only sample files written by the authors were analysed. **Availability.** The source code is released under the MIT licence and is publicly available to review; the measurement script accompanies this paper.

REFERENCES

1. B. Chess and G. McGraw, "Static analysis for security," *IEEE Security Privacy*, vol. 2, no. 6, pp. 76–79, 2004.
2. B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, 2013, pp. 672–681.
3. X. Yin, C. Ni, and S. Wang, "Multitask-based evaluation of open-source LLM on software vulnerability," *IEEE Trans. Softw. Eng.*, 2024, doi: 10.1109/TSE.2024.3470333.
4. O. Çetin, E. Ekmekcioglu, B. Arief, and J. C. H. Castro, "An empirical evaluation of large language models in static code analysis for PHP vulnerability detection," *J. Univers. Comput. Sci.*, 2024, doi: 10.3897/jucs.134739.
5. Yildiz, S. G. Teo, Y. Lou, Y. Feng, C. Wang, and D. M. Divakaran, "Benchmarking LLMs and LLM-based agents in practical vulnerability detection for code repositories," in *Proc. Annu. Meeting Assoc. Comput. Linguistics (ACL)*, 2025, doi: 10.48550/arXiv.2503.03586.
6. Y. Ding *et al.*, "Vulnerability detection with code language models: How far are we?" in *Proc. IEEE/ACM 47th Int. Conf. Softw. Eng. (ICSE)*, 2025, arXiv:2403.18624.
7. L. Zhang, Q. Zou, A. Singhal, X. Sun, and P. Liu, "Evaluating large language models for real-world vulnerability repair in C/C++ code," in *Proc. 10th ACM Int. Workshop Security Privacy Analytics (IWSPA)*, 2024, doi: 10.1145/3643651.3659892.
8. Y. Wu *et al.*, "How effective are neural networks for fixing security vulnerabilities," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, 2023, doi: 10.1145/3597926.3598135.
9. H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in *Proc. IEEE Symp. Security Privacy (SP)*, 2022.
10. Semgrep, "Semgrep: Lightweight static analysis for many languages." [Online]. Available: https://semgrep.dev
11. MITRE, "Common Weakness Enumeration (CWE)." [Online]. Available: https://cwe.mitre.org
12. OWASP Foundation, "OWASP Top 10." [Online]. Available: https://owasp.org/www-project-top-ten/
13. Z. Li *et al.*, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. Netw. Distrib. Syst. Security Symp. (NDSS)*, 2018.
14. Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2019.
15. J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in *Proc. 17th Int. Conf. Mining Softw. Repositories (MSR)*, 2020, pp. 508–512.
16. G. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software," in *Proc. 17th Int. Conf. Predictive Models Data Anal. Softw. Eng. (PROMISE)*, 2021, pp. 30–39.
17. D. Guo *et al.*, "DeepSeek-Coder: When the large language model meets programming—the rise of code intelligence," arXiv:2401.14196, 2024.
18. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 7, October 2026
Impact Factor: 8.2

19. H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *Proc. IEEE Symp. Security Privacy (SP)*, 2023.
20. M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "VulRepair: A T5-based automated software vulnerability repair," in *Proc. ACM ESEC/FSE*, 2022.
21. J. Gajjar, K. Subramaniakuppusamy, R. Puthal, and K. Ranaware, "SecureFixAgent: A hybrid LLM agent for automated Python static vulnerability repair," in *Proc. Int. Conf. Mach. Learn. Appl. (ICMLA)*, 2025, doi: 10.1109/ICMLA66185.2025.00212.
22. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, 2017.
23. M. Xiong *et al.*, "Can LLMs express their uncertainty? An empirical evaluation of confidence elicitation in LLMs," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2024.
24. "Tree-sitter: An incremental parsing system for programming tools." [Online]. Available: https://tree-sitter.github.io
25. "Ollama." [Online]. Available: https://ollama.com
26. NIST, "National Vulnerability Database." [Online]. Available: https://nvd.nist.gov
27. T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, 2012.
28. F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Security Privacy (SP)*, 2014, pp. 590–604.