Exposing Secure Coding Vulnerabilities with a Custom Static Analysis Tool

Anonymous Author(s)

Abstract

Large open-source C and C++ software has suffered from common security and memory-safety vulnerabilities, even with routine code reviews and the reliance of other standard secure coding practices that have been in place. These weaknesses have persisted due to errors in pointer handling, boundary checks, dynamic memory usage, and other classifications categorized by the Common Weakness Enumeration (CWE). This project investigated why these issues have frequently existed in codebases by analyzing widely used repositories and finding common patterns of insecure behaviors that weaken the reliability and security of software.

The team developed a tool with the goal of detecting common security mistakes in C and C++ codebases. The results showed that through static code analysis and the rule-based tool that was built, many issues were found that had been overlooked in their own development and review processes. Suggested by these findings, using these tools as a step in the development pipeline can greatly reduce recurring vulnerabilities in open-source software, and encourage more secure coding practices among developers.

Keywords

Buffer overflow detection, C language, C++ language, Code remediation, Common Weakness Enumeration, Coverity, Memory safety errors, Null pointer de-reference, Open-source, Secure coding practices, Static analysis, Vulnerability discovery

1 Overview

Open-source software (OSS) forms the backbone of many critical systems, including databases, networking platforms, and developer tools. As dependency in OSS has increased, so has the risk posed by hidden or undetected vulnerabilities. Although secure coding guidelines are well established, they have not always been applied consistently across projects, leading to leaks and many forms of defects that slip past code reviews. Addressing these vulnerabilities requires a systematic approach for identifying, classifying, and solving the issues rather than simply detecting them. This study aims to investigate common security weaknesses in large C and C++ open-source projects and to develop practical methods for mitigating them.

Static analysis is used as the primary method for the vulnerability detection such as Coverity Scan being used to select large-scale repositories. Detected issues are categorized using the Common Weakness Enumeration (CWE) system, enabling quantification of recurring vulnerability patterns, such as out-of-bounds memory access, NULL pointer dereferences, and improper memory handling. Analysis reveals that some security flaws exist despite regular code reviews, emphasizing that critical issues often get overlooked during the review process [7]. Large-scale studies have also demonstrated that many vulnerabilities in OSS remained undetected for long periods, highlighting the importance of proactive static analysis to shorten the vulnerability life cycle [14].

Initially, reported defects form Coverity Scan across the selected repositories were collected and categorized. Each defect was mapped to its corresponding Common Weakness Enumeration (CWE) identifier, producing a vulnerability classification table that highlights common categories such as CWE-125 (Out-of-bounds Read), CWE-476 (NULL Pointer Dereference), and CWE-787 (Out-of-bounds Write). This phase allowed quantification of recurring vulnerability patterns in different codebases. Studies by Charoenwet et al. (2024) [7] indicate that such vulnerabilities, particularly memory and resource management issues, are often under-emphasized during standard peer reviews, despite their significant security impact. Aligning each identified issue with its CWE category helped prioritization of vulnerability classes requiring immediate attention.

We then compare the vulnerabilities scanned within the open-source repository Neovim against our tool. Specific defects flagged by Coverity are analyzed to determine their origins, identifying most error-prone coding constructs and examine factors contributed to their persistence in the codebase. These findings provide the foundation for building a custom vulnerability scanning tool capable of identifying basic C and C++ secure coding issues beyond what Coverity automatically detects. The tool aims to perform lightweight static checks for common problems such as uninitialized variables, missing boundary checks, and unsafe function usage. This ongoing development aligns with our broader objective of strengthening vulnerability detection coverage, particularly for smaller or emerging OSS projects that may not have access to enterprise-grade tools like Coverity. By combining empirical observations from established repositories with custom tool design, this phase bridges theoretical vulnerability analysis and practical detection.

The scope of this project is deliberately limited, as the work focuses exclusively on static code analysis of C and C++ projects, and on evaluation of the vulnerability scanner outputs, including the detected CWE classifications, severity levels, and related findings. The project does not cover dynamic analysis, fuzz testing, or full audits of repositories. Instead, the goal is to deliver a tool that can analyze secure coding principles within C++ codebases rapidly. Recent research warns that rapid AI-assisted coding can increase risks [4], making strong security practices more important than ever. By combining Coverity's structured analysis with CWE-based classification and targeted fixes, this project works to improve the security of widely used software and share lessons that support better secure coding practices overall.

Section 2 then explores related studies on vulnerability lifecycles, human factors in code review, and other recent advances in automated vulnerability detection. Following this, section 3 provides an in depth explanation of how repositories were analyzed in regards to vulnerability detection and the development of a lightweight vulnerability scanner to add to the finding from Coverity.

In section 4, we evaluate the effectiveness of our tool vs. Coverity. Finally, sections 5 and 6 go over the ACM Code of Ethics principles that helped guide our approach. The report concludes with section 7 containing a summary of what was learned and the work can help with improving secure coding practices.

2 Related Work

Secure coding vulnerabilities can be introduced into an open-source project in many different ways such as gaining unauthorized access or by stealing sensitive data. By studying these vulnerabilities, the project aims to understand the technical errors and reasons they were able to slip through the standard review and testing processes.

Research on software security has addressed several dimensions, including the lifecycle of vulnerabilities, human factors in code review, automated detection methods, and the impact of emerging AI tools.Understanding how vulnerabilities are introduced, exist and resolved in open-source software (OSS) has been a central focus. Iannone et al. (2023) [11] studied many open-source projects to track when security flaws are added, how long they remain in the codebase, and what factors affect when they are found and how they are repaired. Their study showed that many vulnerabilities stay in software for a long time and are often fixed only after they become public or someone outside the project reports them.Their findings demonstrated that many vulnerabilities persisted for long periods and were often fixed only after public disclosure or external reporting, highlighting that security problems arise not only from coding errors but also from weaknesses in project processes and limited visibility.

Measurement challenges in vulnerability studies were emphasized by Muegge and Murshed (2018) [12], who noted that inconsistent recording and labeling of security fixes across projects could lead to inaccurate results. They recommended clearly defining when a vulnerability starts and when it is fixed, and checking where the data comes from to ensure accuracy.Together, these works illustrate the importance of building carefully curated datasets and applying rigorous analysis to understand vulnerability lifecycles.

Human factors are a major reason why some vulnerabilities persist in open-source software. Code reviewers often focus on functionality, readability, or style rather than security, which allows critical flaws to escape detection. Charoenwet et al. (2024) [7] studied major OSS projects and found that serious security issues, such as buffer overflows, null pointer dereferences, and resource mismanagement, are often under-discussed during reviews. Even when reviewers identify a security problem, disagreements, unclear ownership, or lack of security expertise can delay the implementation of a fix. Complementing this, Alqahtani (2025) [5] highlighted that developers' security awareness and practices vary significantly across projects. This work showed that even experienced contributors may overlook risks or fail to apply secure coding practices consistently. Both studies suggest that human vigilance alone is not enough, and that developers need structured guidance, training, and support tools to improve the likelihood of catching vulnerabilities early. These insights motivated our Phase II work to carefully study repository-specific weaknesses and create a lightweight scanning tool to catch common C and C++ vulnerabilities that might otherwise be missed during regular reviews.

Automated vulnerability detection and classification have also received considerable attention. Zakharov and Gladkikh (2024) [14] analyzed zero-day vulnerabilities in open-source software, examining trends in how such unknown flaws are introduced and remain hidden. Their findings highlight the limitations of existing static and dynamic analysis tools, reinforcing the need for stronger automation and continuous monitoring, which has been explored further through recent advances in AI-assisted security.Complementing this, Ahi and Valizadeh (2025) [4] provided a timely survey on the dual-use nature of large language models (LLMs) and generative AI in cybersecurity. They discussed both offensive uses, such as AI-generated malware or exploit generation, and defensive applications, including intelligent vulnerability detection, explainable security recommendations, and automated patch generation. Their work bridges traditional software security research with emerging AI-driven approaches, offering new perspectives on tool-assisted vulnerability detection, that complements the technical challenges outlined by Zakharov [14] and the process gaps identified by Charoenwet [7] and Iannone [11].

Taken together, these studies provide a comprehensive view of the current software security landscape. They show how important it is to measure vulnerabilities correctly (Muegge & Murshed, 2018) [12], understand why some defects stay in the code for a long time, and how developer behavior affects code reviews (Iannone et al., 2023) [11], (Charoenwet et al., 2024) [7] and (Alqahtani, 2025) [5]. They also show patterns in zero-day vulnerabilities (Zakharov & Gladkikh, 2024) [14] and how AI tools can both help and cause new security problems (Ahi & Valizadeh, 2025)[4]. Overall, these studies tell us that handling vulnerabilities well needs both automated tools and careful human review. This is why research is moving towards combining real data, developer awareness, and smart tools to find and fix problems before they cause trouble.

3 Design and Implementation

This section describes in detail how we designed and implemented our custom static vulnerability scanner for C/C++ code. We explain how rules are represented in TOML, how they are loaded using a C++ TOML parser, how files are discovered and analyzed, and how results are aggregated and reported.

3.1 Vulnerability Selection and Rule Modeling

The scanner is driven entirely by configuration rather than hard-coded checks. Vulnerability patterns are expressed as rules in external TOML files, and the C++ program is essentially a generic engine that loads these rules, applies them to source code, and reports any matches.

Each rule in the TOML configuration describes a single vulnerability pattern. The rule has a name that is used in reports, a short description explaining why the pattern is dangerous, a CWE identifier as a link to a standardized weakness class (for example, `CWE-119` for buffer overflows), and a qualitative severity such as `low`, `medium`, `high`, or `critical`. The rule also contains one or more regular expressions that describe syntactic patterns in C/C++ code, and a short remediation recommendation that suggests safer APIs or defensive programming practices.

In practice, the rule files cover common security issues that can be recognized through textual patterns: unsafe buffer operations such as unbounded `strcpy` or `gets` calls (CWE-119), uncontrolled format strings in `printf`-like functions (CWE-134), shell command execution through `system`, `popen`, or `exec*` with untrusted input (CWE-78), hard-coded credentials or secrets embedded as string literals (CWE-798), and simple patterns indicative of null pointer dereferences (CWE-476). The important point for the implementation is that the C++ code does not need to know anything about these categories, only needing to understand that a rule has metadata and a collection of regular expressions to apply.

At startup, the TOML files are parsed into an internal representation where each rule is stored as a C++ object containing the name, CWE, severity, description, recommendation, and a vector of compiled `std::regex` objects corresponding to the patterns defined in TOML. This translation step is responsible for bridging the declarative TOML description and the executable matching logic.

3.2 Core Scanner Architecture

The scanner is implemented in C++ and organized as a pipeline that starts from command-line options, loads rules from TOML using the toml11 library, discovers files in a given directory tree, scans each file line by line with the compiled rules, and finally collates and reports the findings.

The program begins in `main` by parsing command-line arguments that specify at least the root directory to scan and the path to a TOML rule file. Additional options control which source file extensions are included (for example, `.c`, `.cpp`, and `.h`), which directories or files should be excluded (such as build directories or third-party dependencies), and what format should be used for the output. These options are stored in a configuration structure that is passed to the subsequent components, ensuring that all scanning decisions are governed by the same settings.

Once configuration is established, the scanner loads the vulnerability rules from the specified TOML file. This is done using the header-only toml11 library, which parses the TOML into a `toml::value` tree. The code then navigates this tree, iterating over each rule table and extracting fields such as the rule name, description, CWE string, severity, recommendation, and the array of pattern strings. For each TOML rule, the scanner constructs a corresponding C++ rule object and compiles each pattern string into a `std::regex`. If any required field is missing or a regular expression cannot be compiled, the scanner either reports an error and aborts or skips the faulty rule, depending on how strict the behavior is configured. After this stage, the scanner has an in-memory vector of rules that are completely detached from the TOML representation and ready for efficient application to source code.

File discovery is implemented using the standard C++17 filesystem library. Given the root directory, the scanner traverse the entire tree directory tree recursively. As it encounters each path, it checks whether the path is a directory that should be excluded based on the configuration. For regular files, it examines the file extension and keeps only those that correspond to C/C++ sources or headers. This yields a set of candidate files that will be analyzed. Because file discovery is isolated from the rule engine, it can be modified

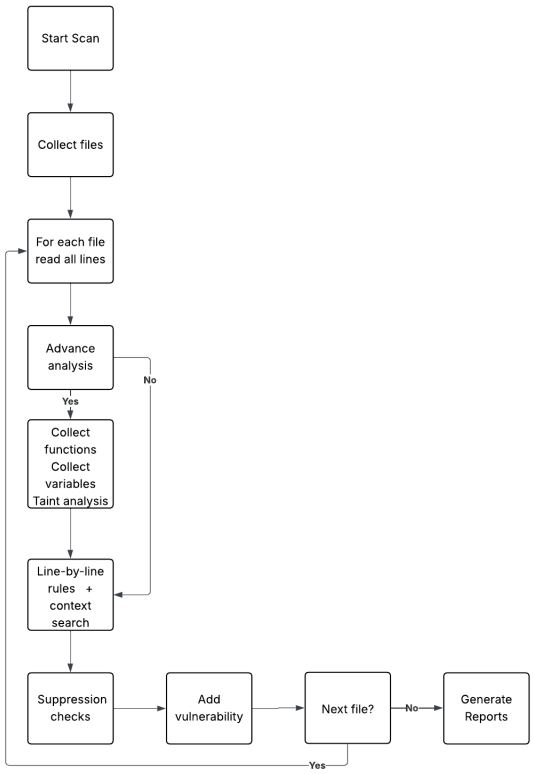

Figure 1: Flow chart showing the main steps and data flow in the project.

(for example, to add or remove extensions) without changing any of the vulnerability logic.

Analysis itself is line-oriented. For each selected file, the scanner opens it as a text stream and reads one line at a time while maintaining a line number counter. Before applying rules, it performs simple preprocessing: empty lines are ignored, and lines that are clearly comments (such as lines starting with `//`) can be skipped entirely. The implementation can also track whether it is inside a block comment delimited by `/*` and `*/`, so that rule matching is not performed on commented-out code. This avoids a large class of trivial false positives.

For each nontrivial line, the scanner iterates over all loaded rules and their associated regular expressions. For a given rule, it applies each `std::regex` to the line using `std::regex_search`. If any pattern matches, the scanner constructs a finding object that records the file path, the line number, the line of code as a snippet, and the rule metadata such as name, CWE, severity, description, and recommendation. This finding is appended to a vector of findings that persists for the duration of the scan. The engine does not attempt to deduplicate findings across rules.

Some rules provide very simple contextual checks by encoding them as additional regular expressions or by relying on the structure of the pattern itself. For instance, a format-string rule might

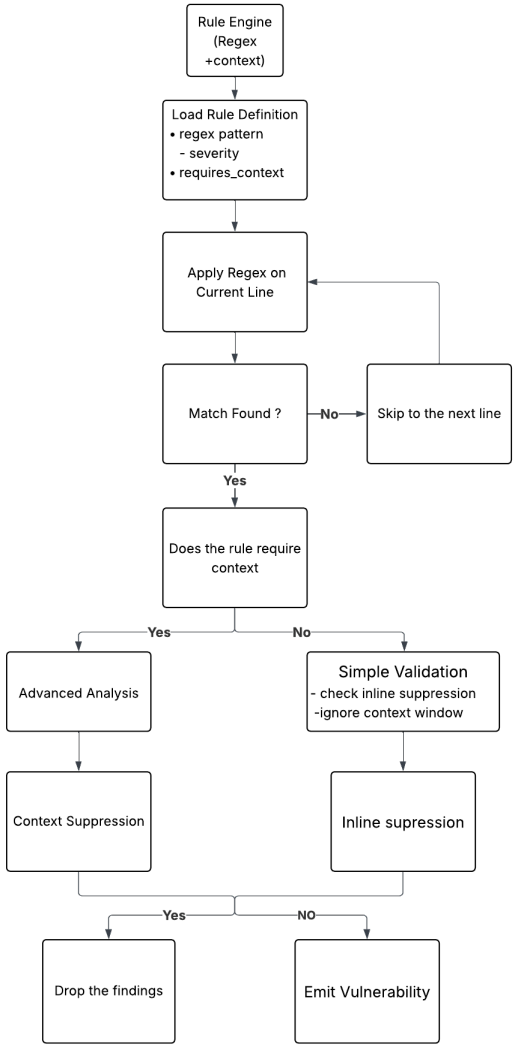

Figure 2: Rule Engine Diagram: Shows how each code line is matched against regex rules and optionally analyzed

use a pattern that matches calls to `printf` where the first argument does not appear to be a quoted string literal, distinguishing obvious safe cases like `printf("%s", user_input);` from more suspicious ones like `printf(user_input);`. Similarly, rules for buffer overflows employ word boundaries in their regular expressions so that the scanner matches `strcpy(` but not identifiers such as `strcpy_wrapper`. Although these checks are still purely syntactic and local to a single line, they substantially reduce the noise compared to blindly searching for substrings.

After all files have been processed, the scanner aggregates the findings and produces the final report. Internally, aggregation is straightforward: the vector of findings is traversed once to compute summary statistics such as the total number of files scanned, the total number of findings, and the distribution of findings per severity level. The tool then prints a concise summary followed by a detailed listing where each finding is reported with its file, line number, rule name, CWE identifier, severity, code snippet, and recommended remediation.

3.3 Design Trade-offs and Extensibility

Using TOML and `toml11` for rule configuration leads to a clear separation between the analysis engine and vulnerability knowledge:

- **Pros**:
  - New vulnerability types or variants can be added by editing TOML files, with no changes to the C++ codebase.
  - Rules are human-readable and version-controllable, making them suitable for collaborative refinement.
  - The C++ core remains relatively small and focused on file traversal, regex application, and reporting.
- **Cons**:
  - The scanner is limited to patterns expressible as regular expressions on individual lines (or small windows of text).
  - There is no AST-level or inter-procedural analysis, so the tool cannot reason about control flow, data flow, or pointer aliasing.

Compared to heavyweight static analyzers such as Coverity, our TOML-driven C++ implementation is intentionally lightweight. We demonstrate how a configurable rule engine, built on top of a standard filesystem API, `std::regex`, and `toml11`, can automatically surface a meaningful subset of common security issues in C/C++ projects.

4 Analysis

We apply our TOML-driven vulnerability scanner to the opensource text editor Neovim, written in C++. We then use this output to compare and contrast against Coverity scan's analysis report on the same repository. Our results demonstrate that while Coverity produces analysis that access to larger context, the combined use of both tools shows a layered approach to secure coding that enables safer verification of C and C++ codebases.

Coverity Scan and similar tools perform deep semantic and static analysis. This type of analysis enables the same analysis techniques that compilers use within optimizations. For instance, Coverity uses a data-flow analysis pass combined with memory simulations. This enables their tool to identify difficult to determine problems, such as dead code (which could be exploited by an attacker) and integer overflows. They are also able to track pointers that may be doubly free'd, or not free'd at all, via their data flow modeling.

Conversely, our tool functions on syntactic analysis. While it covers a narrower scope, only detecting 11.8% of the vulnerability categories found by Coverity, it has a clear purpose functioning as a secure coding principles compliance checker. With the ability to define syntactic rules and warn on their existence, a programmer must justify their use of potentially insecure practices, for instance `strcpy` vs. `strncpy`. This enforces a coding style that is enforces security from the beginning, preventing technical debt from accumulating. Coverity looks for potential crashes, while our tool checks is the codebase compliant with modern secure coding practices.

This analysis shows that our tool is not a competitor to enterprise static analysis tools, but serves as a fast, lightweight, and extendable

tool that can be run prior to larger tools like Coverity Scan. While our tool can be run in seconds on large codebases, Coverity takes hours to index entire repositories, typically running only once per day or once per week. Our tool successfully catches simple issues within insecure functions immediately, which frees up developer time and resources for the deeper, more expensive analysis provided by tools like Coverity.

Table 1: Syntactic Scanner vs. Coverity SAST

Metric	Syntactic Scanner	Coverity Scan
Methodology	**Pattern Matching:** Fast, regex-based rules defined in TOML.	**Semantic Analysis:** Deep inspection of memory paths and logic.
Total Defects	15	134
Categories	2 (Overflows, Leaks)	17 (Integer Errors, Null Deref, Leaks, etc.)
Coverage	**11.8%** of total categories found.	**100%** comprehensive coverage.
Context	**Strict Enforcement:** Flags all banned patterns (e.g., wrappers) to enforce hygiene.	**Heuristic Filtering:** Ignores dangerous patterns if logic suggests safety.
Speed	**High Speed:** Immediate feedback suitable for pre-commit hooks.	**High Latency:** Requires full build integration and processing time.
Role	**Security Linting:** Enforces code hygiene, prevents unsafe primitives.	**Debugger:** Identifies runtime logic errors and crashes.

5 Legal Considerations

In the context of this research, which investigates secure coding vulnerabilities in open-source software, legal considerations form a crucial backdrop to the use of automated scanners, the discovery of weaknesses, and the ethical disclosure of those findings. The legislative frameworks governing cybersecurity, data protection, software licensing, and vulnerability research have evolved significantly in recent years (2022–2023), highlighting the growing recognition of software security as a national and international priority. This section examines the key legislative areas relevant to the detection and remediation of coding vulnerabilities, focusing on existing laws as well as emerging regulatory frameworks shaping modern secure software development..

5.1 Existing Laws

The Computer Fraud and Abuse Act (CFAA) governs unauthorized access in the United States. The Act criminalizes obtaining information from any protected computer without authorization or by exceeding authorized permission [1]. This is particularly relevant for researchers conducting automated vulnerability scans, which if directed at systems without explicit consent can be interpreted as unauthorized probing. Recent academic papers emphasize that automated scanners, like the one developed in this project, must only be employed within controlled environments to avoid CFAA implications. Although this study avoids interaction with external systems, future applications in real open-source software may require legal agreements or responsible disclosure protocols to ensure compliance.

While the code analyzed in this project does not handle personal data, the vulnerabilities detected (buffer overflows, format-string misuse, unsafe input functions) can lead to privacy breaches if present in live systems. The General Data Protection Regulation (GDPR), places strict requirements on software developers and organizations to ensure "appropriate technical and organizational measures" to protect personal information [2]. A buffer overflow that results in memory leakage or a format-string vulnerability that exposes internal variables could constitute a failure to meet GDPR's Article 32 security expectations. Thus, while this project works in an isolated test environment, the nature of the identified vulnerabilities has direct legal relevance for any organization deploying similar code.

Complementing GDPR, the UK Data Protection Act 2018 mandates robust protection of personal information and assigns accountability for breaches resulting from insecure software [3]. Organizations deploying open-source components must ensure those components are free from known security defects, such as the unsafe functions identified in this project's scan. If exploited in real systems, such vulnerabilities can lead to regulatory penalties, breach notifications, and reputational damage. Recent studies emphasis that insecure open-source dependencies have increasingly become a target for regulatory scrutiny, particularly after several high-profile supply chain vulnerabilities emerged in open-source ecosystems.

Open-source software is typically distributed under licenses such as MIT, GPLv3, and Apache 2.0, each imposing specific obligations on modification, analysis, and redistribution. While most open-source licenses allow examination of source code, redistributing modified code (such as patched versions addressing vulnerabilities) requires adherence to licensing terms. For instance, GPL-based projects mandate that derivative works must be published under the same license, while permissive licenses require attribution. In vulnerability research, proper handling of discovered flaws including documentation and responsible disclosure is essential for avoiding IP-related disputes. Academic papers from 2022–2023 emphasize that IP mismanagement remains a major risk when security researchers publish modifications or analyses of open-source software.

5.2 Compliance with Contributor Requirements

Most open-source repositories have a `CONTRIBUTING.md` file which states the requirements and standards for contributing patches. We researched each project's contribution process to understand their legal requirements:

- Many repositories, such as the GNU Compiler Collection [8], require contributors to sign Contributor License Agreements (CLAs) that grant explicit rights to project maintainers.

- Other projects require a Developer Certificate of Origin (DCO) [10] which is signed off by the contributor, asserting that the code was created by them, it complies with the open-source license, and that their code may be redistributed.
- We document our contribution process to demonstrate that we followed each project's specific legal requirements.

5.3 Legal Exposure

We acknowledged that our security contributions come with inherent limitations:

- We provide patches as-is without guaranteeing their effectiveness or completeness.
- We understand that we cannot be held liable for any damages resulting from our vulnerability assessments or proposed fixes.
- We recognize that downstream users must evaluate and test our patches before deployment.

These limitations protect us from any legal liability by submitting potentially incorrect patches.

5.4 Proposed Legislation

The Cyber Resilience Act [9], proposed by the European Commission, is expected to influence software development practices across the EU . It aims to regulate the security of digital products, including open-source components used commercially. The CRA proposes mandatory secure development practices, continuous vulnerability management, and rapid patch deployment for products entering the European market. If adopted, the Act could impose legal obligations on maintainers of open-source projects whose software is integrated into commercial devices. The types of vulnerabilities identified in this project particularly buffer overflows fall under the categories explicitly targeted by the CRA's requirements.

In the United States, the NIST Secure Software Development Framework [13], increasingly referenced in federal procurement policies, outlines mandatory secure coding practices for suppliers of software to government agencies. This proposed expansion would require developers to phase out unsafe functions such as gets and strcpy, mandate vulnerability scanning similar to the tool developed in this project, and enforce documentation of remediation practices. If formalized as federal policy, these requirements would elevate the legal significance of secure coding and vulnerability identification.

The integration of legal considerations into the analysis of secure coding vulnerabilities is essential. As legislation evolves rapidly in response to cybersecurity threats, developers and researchers must remain vigilant, ensuring that vulnerability-scanning activities, remediation strategies, and disclosure processes comply with established and emerging legal frameworks

6 Ethical Considerations

When performing Coverity scans and contributing to open-source material, following the ACM Code of Ethics ensures that work is completed responsibly and with respect to the authors and communities maintaining these projects, and also for those who use the end product. It helps with focusing on improving code security, and in our case, guide how we identify vulnerabilities and produce patches.

6.1 Protecting Users and Ensuring Fair Treatment

Within the domain of secure software development, **Contribute to Society and Human Well-being (Principle 1.1)** emphasizes the responsibility of professionals to ensure that the systems they build or analyze promote the well-being of society [6]. Open-source software plays a foundational role in digital infrastructure, and vulnerabilities such as buffer overflows and unsafe input handling can have far-reaching consequences if exploited. By identifying and documenting these issues, this project contributes to a safer and more secure software ecosystem. The remediation recommendations provided such as replacing unsafe functions and ensuring proper memory management, support developers in producing code that protects users and maintains system integrity.

Adding to treating users in a fair nature in the context of secure coding, **Be Fair and Take Action Not to Discriminate (Principle 1.4)** relates to equitable access to secure and reliable software [6]. Vulnerabilities disproportionately affect users who may depend on open-source components in critical contexts, including those with limited ability to patch or modify code. Ethical practice demands that researchers present findings in a way that does not unfairly target specific projects or maintainers but instead supports the broader community in addressing systemic issues. This also includes ensuring that security tools and documentation are accessible and usable by all developers, regardless of background or experience level

Respecting Privacy (Principle 1.6) is also a key concern when it comes to the protection of users and their fair treatment [6]. Memory-safety vulnerabilities are a huge concern, as they often serve as vectors for privacy violations by enabling exposure of sensitive data. Even though the test file used in this project does not process real data, the underlying ethical principle remains highly relevant. Researchers must consider how similar vulnerabilities behave in real systems and recognize that insecure coding practices can directly threaten user privacy. This principle reinforces the importance of identifying and mitigating insecure patterns, thereby safeguarding the confidentiality of data in real deployments.

6.2 Preventing Harm Through Accurate Security Evaluations

Avoiding harm (Principle 1.2) is central to the ethical practice of cybersecurity research. Vulnerabilities such as those identified in this project pose severe risks if exploited, including unauthorized access, data breaches, or system compromise. This principle underscores the necessity of conducting all scanning and analysis on authorized, controlled files to prevent accidental disruption of external systems [6]. Moreover, it guides the responsibility to avoid disclosing vulnerabilities in ways that could empower malicious actors. The decision to limit testing to test files and not real-world open-source systems reflects a commitment to minimizing harm and ensuring ethical stewardship of security tools.

Aligning with the idea of preventing harm directly, the necessity to **Give Comprehensive and Thorough Evaluations (Principle 2.5)** helps with ensuring no weaknesses are overlooked in order to stay ahead any risks that could potentially cause harm to users in the future. Professionals are ethically obligated to provide thorough

and technically accurate evaluations [6]. This project fulfills that requirement by offering detailed descriptions of each vulnerability, explaining its implications, and recommending specific mitigation strategies. The inclusion of severity levels helps stakeholders prioritize corrective actions. This principle also discourages partial or superficial analysis, ensuring that vulnerability assessments meaningfully contribute to improving system security.

6.3 Responsible Access When Contributing to Secure Systems

When performing any work with open-source repositories or other systems, it is of key importance to ensure you gain proper permissions and **Access Computing Resources Only When Authorized (Principle 2.8)** [6]. This principle aligns directly with legal requirements under the CMA and CFAA. Ethical researchers must ensure that vulnerability scans are conducted only on systems for which explicit authorization has been given. By restricting testing to a local, controlled file created for research purposes, this project adheres to the ethical boundaries defined by the ACM.

Open-source ecosystems also depend on community contributions to maintain security and functionality. By identifying vulnerabilities and providing actionable remediation guidance, along with proper permission, this project contributes to the stability and resilience of widely used software components. **Contribute to the Stability and Security of Computing Ecosystems (Principle 3.1)** reinforces that secure coding research, when conducted responsibly, supports society's reliance on open-source systems [6].

The ethical considerations governing this project extend beyond technical correctness. They encompass a commitment to transparency, fairness, non-maleficence, and respect for user privacy. The project ensures that its research approach aligns with accepted professional norms and contributes positively to the broader field of cybersecurity.

6.4 Other Ethical Aspects

Beyond ACM principles, the project adheres to general ethical norms in cybersecurity and software engineering:

- **Data Security Awareness:** The project educates developers about secure practices, contributing to a culture of security-conscious software development.
- **Promotion of Trustworthy Open-Source Ecosystems:** By improving code security and providing actionable guidance, the project supports the integrity and reliability of open-source software, fostering confidence among users and contributors.

7 Conclusions

This project explored common security vulnerabilities in large open-source software and developed practical methods for detecting and remediating them. We compare the efficacy of tools designed for linting secure coding principles with larger tools like Coverity Scan, demonstrating that both have their place within the software development toolkit. We analyzed the opensource repository Neovim, and found critical vulnerabilities that are still yet to be fixed. These findings show that even in actively maintained and reviewed projects, subtle coding mistakes exist, demonstrating that traditional code review alone is insufficient to ensure secure software development.

To address these vulnerabilities, the project implemented a custom static analysis tool capable of detecting high-risk patterns in C and C++ code. The tool not only identified common flaws such as buffer overflows, format-string vulnerabilities, and memory leaks, but also provided suggested secure coding practices to remediate them. This combination of automated detection and actionable guidance illustrates how layered strategies can significantly reduce recurring vulnerabilities in open-source software.

This project demonstrates that strengthening the security of open-source software requires coordinated technical innovation, legal awareness, and ethical responsibility. By combining automated detection with thoughtful analysis and recovery strategies, the work undertaken here contributes to the broader effort to build safer, more reliable, and more trustworthy software within the open-source community.

References

[1] 1986 (amended). Computer Fraud and Abuse Act, 18U.S.C. §1030. https://www.law.cornell.edu/uscode/text/18/1030. United States Code, Title 18, Chapter 47.

[2] 2016. Regulation (EU) 2016/679 (General Data Protection Regulation). https://gdpr-info.eu/. Official Journal of the European Union, L 119, 04.05.2016.

[3] 2018. Data Protection Act 2018. https://www.legislation.gov.uk/ukpga/2018/12/data.html. UK Public General Acts, c. 12.

[4] Kiarash Ahi and Saeed Valizadeh. 2025. Large Language Models (LLMs) and Generative AI in Cybersecurity and Privacy: A Survey of Dual-Use Risks, AI-Generated Malware, Explainability, and Defensive Strategies. In *2025 Silicon Valley Cybersecurity Conference (SVCC)*. 1–8. doi:10.1109/SVCC65277.2025.11133642

[5] Sultan S. Alqahtani. 2025. Beyond the code: analyzing OSS developers security awareness and practices. *International Journal of Information Security* 24, 3 (2025), 109. doi:10.1007/s10207-025-01023-1

[6] Association for Computing Machinery. 2018. ACM Code of Ethics and Professional Conduct. ACM, New York. https://www.acm.org/code-of-ethics.

[7] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. 2024. Toward effective secure code reviews: an empirical study of security-related coding weaknesses. *Empirical Software Engineering* 29, 4 (2024), 88. doi:10.1007/s10664-024-10496-y

[8] GCC Steering Committee. 2021. Update to GCC Copyright Assignment Policy. GCC mailing list announcement. Contributors may use a Developer Certificate of Origin instead of assigning copyright to FSF.

[9] European Parliament and Council of the European Union. 2024. Regulation (EU) 2024/2847 of the European Parliament and of the Council on horizontal cybersecurity requirements for products with digital elements (Cyber Resilience Act). https://eur-lex.europa.eu/eli/reg/2024/2847/oj. OJL2847, 20Nov2024; enters into force 11Dec2027..

[10] Linux Foundation and its contributors. 2006. Developer Certificate of Origin, Version1.1. Public legal text. Used in many open-source projects as an alternative to CLA.

[11] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2023. The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study. *IEEE Transactions on Software Engineering* 49, 1 (2023), 44–63. doi:10.1109/TSE.2022.3140868

[12] Steven M. Muegge and S. M. Monzur Murshed. 2018. Time to Discover and Fix Software Vulnerabilities in Open Source Software Projects: Notes on Measurement and Data Availability. In *2018 Portland International Conference on Management of Engineering and Technology (PICMET)*. 1–10. doi:10.23919/PICMET.2018.8481833

[13] Karen Scarfone, Murugiah Souppaya, and Donna Dodson. 2022. *Secure Software Development Framework (SSDF) Version1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities.* Technical Report Special Publication NIST SP800-218. National Institute of Standards and Technology (NIST), U.S. Department of Commerce. doi:10.6028/NIST.SP.800-218

[14] Alexander A. Zakharov and Kirill I. Gladkikh. 2024. Characteristics and Trends of Zero-Day Vulnerabilities in Open-Source Code. In *2024 International Russian Automation Conference (RusAutoCon)*. 498–502. doi:10.1109/RusAutoCon61949.2024.10694228