Author's copy of the paper published in Proceedings of the IEEE 5th Cyber Awareness and Research Symposium 2025 (CARS'25). Grand Forks, ND, October 27- 30, 2025.

Empirical Analysis of Security Vulnerabilities in Open Source Software Using Static Analysis Tools

Donghoon Kim
Department of Computer Science
Arkansas State University
Jonesboro, AR, USA
dhkim@astate.edu

Hokeun Kim
School of Computing and Augmented Intelligence
Arizona State University
Tempe, AZ, USA
hokeun@asu.edu

***Abstract*—Open-source software (OSS) is vital to modern ecosystems but often lacks rigorous security auditing found in proprietary systems. Despite collaboration and static analysis, many OSS projects still include undetected vulnerabilities. Limited empirical research on the persistence of security issues hinders efforts to improve detection and maintain the security of OSS. To address this problem, we empirically investigate vulnerabilities in real OSS projects by applying the static analysis tool SonarQube, complemented with manual validation and case studies to assess identifiable security flaws in the code and the false positives generated during analysis. The results show that larger projects face more security vulnerabilities due to complexity, while maintainability and reliability depend on architecture and code quality than on size. Static analysis tools can detect straightforward flaws such as hardcoded credentials and weak cryptography but often fail to capture complex or context-specific issues. The study also reveals significant false positive rates, highlighting the need for manual validation. This study reveals security flaws in open-source software and evaluates static analysis tools, stressing the need for better awareness and context-aware practices.**

***Index Terms*—Open source software, security vulnerability, static analysis tool, empirical study**

I. Introduction

Open source software (OSS) forms the backbone of today's software infrastructure, powering systems ranging from operating systems and web frameworks to cryptographic libraries and data analytics platforms [1]. The collaborative and transparent nature of OSS development enables rapid innovation, but it also introduces substantial challenges for ensuring security, particularly in the absence of centralized auditing or uniform development standards [2]. In recent years, there has been a significant increase in the development of automated tools for detecting vulnerabilities in open-source software (OSS) [3]. Among these, static analysis tools such as SonarQube [4], Checkstyle [5], and FindBugs [6] have gained widespread adoption due to their ease of integration into development pipelines, support for multiple programming languages, and extensive rule sets for identifying bugs, code smells, and specific classes of security issues [7]. However, there is a notable lack of empirical studies that systematically examine the extent to which real-world OSS projects remain exposed to security vulnerabilities despite the use of such tools [8], [9]. Furthermore, there is a continued need to investigate the inherent limitations of these tools in practice, particularly in terms of their ability to detect subtle or context-specific vulnerabilities, as well as their potential to generate false positives that may diminish developer trust and reduce remediation effectiveness [10].

Given the limited empirical understanding of the performance of static analysis tools in real-world development environments, this study investigates the following research questions. (RQ1) To what extent do actively maintained open-source software (OSS) projects remain exposed to security vulnerabilities despite the use of widely adopted static analysis tools? (RQ2) What types of limitations, such as the omission of context-specific vulnerabilities or a high rate of false positives, undermine the practical effectiveness of these tools in supporting secure software development? By addressing these questions, our study aims to assess the actual security exposure of OSS projects even when static analysis tools are in place, and to identify critical gaps that may compromise the reliability and practical utility of these tools in safeguarding the broader OSS ecosystem.

The purpose of this research is twofold. First, we aim to conduct a systematic investigation of security vulnerabilities present in a curated set of actively maintained open-source software (OSS) projects by leveraging SonarQube as our primary static analysis tool. Through this process, we seek to quantify and characterize the types of vulnerabilities that surface in real-world OSS codebases, even when widely adopted automated tools are integrated into their development pipelines. Second, we perform an in-depth analysis of SonarQube's outputs to identify and categorize its diagnostic strengths and weaknesses, with particular attention to classes of vulnerabilities that are either over-reported (i.e., false positives that may overwhelm or mislead developers) or under-reported (i.e., false negatives that allow security risks to persist undetected). In addition, we evaluate the suggested fixes from the static analysis, following the suggestions and rerunning the static analysis to confirm the benefits and limits of these features. Rather than aiming to evaluate the accuracy or performance of SonarQube in isolation, we employ it as a representative case to gain insight into the broader capabilities and limitations of static analysis tools when applied in practice. By using SonarQube as an

analytical lens, this study contributes to a deeper empirical understanding of the current state of automated vulnerability detection in OSS, exposing where such tools fall short and how those shortcomings may impact secure development practices in the open-source ecosystem.

To conduct our analysis, we select a diverse set of actively maintained open-source projects from the OpenHub [11] platform. These projects are evaluated, focusing on a range of quality and security-related metrics, including security vulnerabilities, reliability issues, maintainability concerns, and security hotspots as defined by SonarQube. A comprehensive assessment is performed to understand the overall security posture of each project. In addition, we conduct a case study analysis to examine whether these projects exhibit practical security issues in real-world contexts. This case study further allows us to highlight the limitations of SonarQube, particularly in its ability to detect context-specific vulnerabilities, and to illustrate instances of false positives that may mislead developers or reduce the actionable value of the tool's output. This study makes the following contributions.

- **Empirical Assessment of OSS Security Exposure**: We conduct a systematic evaluation of security vulnerabilities in actively maintained open-source software projects using real-world codebases. This analysis provides concrete evidence of the extent to which security issues persist despite the integration of widely adopted static analysis tools into modern development workflows.
- **Characterization and Limitations of Static Analysis Tools**: Through comprehensive analysis and targeted case studies, we identify key limitations of static analysis tools, such as their tendency to produce false positives and miss context-dependent vulnerabilities. These findings underscore the practical challenges developers encounter when depending on automated approaches for security auditing in real-world software projects.

The structure of this paper is as follows. Section II reviews relevant literature on security in OSS projects and the limitations of static analysis tools. Section III outlines our research methodology, including project selection, SonarQube configuration, and the manual validation process. Section IV presents the quantitative results of our analysis, summarizing the distribution and accuracy of vulnerabilities detected across multiple OSS projects. Section V provides detailed case studies that highlight key strengths and limitations of SonarQube, including examples of false positives. Finally, Section VI concludes this paper by summarizing our findings and suggesting directions for future research.

II. Related Work and Background

A. Security in Open Source Software

The security of open source software (OSS) has been the subject of significant research. Studies have identified that OSS projects vary widely in terms of secure coding practices, peer review rigor, and automated testing. Researchers have also highlighted the increasing reliance on OSS components in both commercial products and critical infrastructure. This widespread adoption significantly amplifies the potential impact of even small, undetected vulnerabilities [12], [13]. A seemingly minor flaw in a commonly used OSS library, such as an unchecked input or an insecure default configuration, can serve as an entry point for attackers when the library is integrated into larger systems. Since OSS packages are frequently reused across thousands of dependent projects, a single overlooked vulnerability can propagate through the software ecosystem and lead to widespread security risks.

Open-source project hosting platforms such as GitHub [14] provide baseline security features aimed at monitoring code changes and automatically detecting known vulnerabilities [15]. These features include dependency scanning tools, notification systems for published CVEs, and automated patch recommendations. However, recent empirical studies [16], [17] have shown that these security mechanisms are frequently misconfigured, improperly used, or altogether disregarded. Consequently, developers may operate under the false assumption that their projects are secure, while latent vulnerabilities persist. This misplaced confidence fosters a false sense of protection and can undermine the adoption of more proactive and comprehensive security practices. Despite the critical importance of these issues, empirical research on the nature, prevalence, and evolution of security vulnerabilities in open-source software remains limited, particularly with respect to how these vulnerabilities are surfaced, interpreted, or overlooked in real-world development workflows [2].

B. Limitations of Static Analysis Tools

Static analysis tools have become integral to modern software development workflows, offering a scalable and automated means of enforcing code quality and identifying vulnerabilities early in the development lifecycle. A substantial body of academic and industrial research has assessed these tools across various dimensions, such as detection accuracy, false positive rates, developer usability, and applicability across languages and domains [18], [19]. These studies highlight both the promise and the practical challenges of integrating static analysis into secure software engineering practices. Among widely adopted tools, Coverity [20], CodeQL [21], and SonarQube [4] have demonstrated the ability to detect a range of software vulnerabilities, including insecure cryptographic configurations, buffer overflows, and improper input validation. These tools are particularly effective at identifying issues that follow recognizable syntactic or semantic patterns.

From a security standpoint, static analysis tools are limited in their ability to capture data- and control-flow interactions across complex software structures, such as interprocedural data flows, asynchronous callbacks, deeply nested function calls, and polymorphic method dispatch [22], [23]. These limitations hinder the detection of security issues like privilege escalation, access

control bypasses, and tainted data propagation across multiple layers of abstraction [24]. Additionally, static analysis often produces a high number of false positives, particularly when analyzing large or modular codebases, which can overwhelm developers and reduce the likelihood of actionable remediation [22], [25].

Several studies have examined the performance and limitations of static analysis tools. Bessey *et al.* [23] explored real-world deployments of Coverity [20] and observed that many reported warnings are not actionable due to high false positive rates. Similarly, Johnson *et al.* [26] studied developer responses to static analysis warnings and found that warning fatigue and lack of context reduced developer trust in such tools. Building on these findings, recent evaluations have examined other widely adopted tools such as SonarQube. Its integration with development workflows and its broad rule set make it appealing for large-scale quality assurance. However, prior evaluations [10] have found inconsistencies in its vulnerability reporting, particularly around rule precision and CWE alignment.

C. Open Source Projects Used for Our Study

This study leverages SonarQube as a practical lens to investigate vulnerabilities in actively maintained real-world OSS projects to perform hands-on, in-depth case study analysis.

Specifically, we target two open-source projects that we regularly contribute to, and conduct in-depth case studies to qualitatively evaluate the effectiveness of SonarQube in practice. These two open-source projects are (1) Secure Swarm Toolkit (SST) [27] and its C API [28], [29] which we call `SST C API` in this paper, and (2) Lingua Franca (LF) [30], [31] and its Reactor-C repository [32] which we call `Lingua Franca Reactor-C` or `LF Reactor-C` in short in this paper. Specifically, we follow the suggestions for fixing vulnerabilities by SonarQube on these open-source repositories and validate if vulnerabilities can be fixed and confirmed by SonarQube. Details of case studies on these open-source projects are discussed in Section V.

III. Methodology

To ensure a comprehensive and reliable evaluation, we design a systematic workflow that includes careful project selection, rigorous static analysis configuration, and thorough manual validation of findings. We first identify representative open-source projects that meet specific criteria to guarantee relevance and analytical depth. Next, we employ SonarQube for automated static analysis and complement these results with meticulous manual reviews to confirm the accuracy of identified issues.

A. Project Selection

We select a diverse and representative set of open-source software projects from the OpenHub platform [11] to ensure the robustness and generalizability of our analysis. The project selection process is guided by a set of well-defined criteria designed to capture both technical relevance and practical feasibility:

- **Programming Language**: We focus on projects implemented in C, C++, Java, or Python, as these languages are among the most widely used and influential in the current software ecosystem, as reflected in the TIOBE Programming Community index [33]. In particular, we deliberately prioritize projects predominantly written in C and C++, since these languages are extensively utilized for system-level development and are directly related to system security. Their low-level memory management and manual control over resources introduce distinctive security challenges, making them especially relevant for evaluating vulnerability patterns and secure coding practices [34].
- **Activity**: To guarantee that the projects are actively maintained and receive ongoing contributions, we include only those with recent commits and an active contributor base, defined as having at least five current contributors. This criterion helps ensure that the selected projects are not stagnant or abandoned, which could otherwise bias the security and quality assessments.
- **LOC (Lines of Code)**: We consider project scale by including only those with at least 10,000 Physical LOC, thereby focusing on non-trivial codebases that pose realistic maintenance and security challenges. At the same time, excessively large projects are excluded to prevent analysis failures or impractical processing times during static analysis.

Based on these criteria, we ultimately select 20 projects, as summarized in Table I, spanning a diverse range of application domains such as system libraries, web frameworks, and utility tools. This selection provides a comprehensive and representative foundation for our subsequent evaluation.

B. SonarQube Configuration and Analysis

For our analysis, we use `SonarQube cloud` [4], a cloud-based service of SonarQube. SonarQube integrates seamlessly with popular version control platforms such as GitHub, Bitbucket, GitLab, and Azure DevOps, enabling efficient and automated static code analysis workflows. In this study, SonarQube is configured to import projects directly from GitHub repositories. To facilitate this process, the user first forks the target repository into their personal GitHub account, thereby creating an independent copy that can be freely analyzed without affecting the original source. The forked repository is then linked to SonarQube, allowing the platform to automatically initiate static analysis workflows upon connection. Once integrated, SonarQube clones the entire source code and conducts a comprehensive evaluation using its advanced rule-based analysis engine. This analysis identifies various types of issues, including security vulnerabilities, reliability problems, maintainability concerns, and security hotspots. By leveraging SonarQube's seamless integration capabilities and

TABLE I: Summary of Open-Source Project Vulnerabilities Detected by SonarQube

Project	LOC	Contributors	Security	Reliability	Maintainability	Security Hotspots
Kamailio	1.1M	542	27	190	29,000	3,400
cURL	202K	1,474	308	45	4,600	1,100
Linux Test Project (LTP)	403K	579	60	709	15,000	2,100
Neat Project	30K	39	2	73	1,200	84
PROJ (Cartographic Library)	271K	215	1	334	15,000	275
Swift Corelibs Foundation	271K	531	3	137	8,600	82
Automotive Grade Linux	41K	195	0	2,300	2,900	3
GRASS GIS Addons	307K	171	13	946	15,000	918
Unity Test	19K	171	0	10	458	5
Lingua Franca (LF) Reactor-C	20K	36	0	31	688	56
SST C API	4.4K	10	1	18	209	53
Snort3	355K	38	2	171	30,000	599
Zephyr	2.3M	2,800	4	1,700	66	1,700
Kdenlive	214K	200	0	95	11,000	11
dlib C++ Library	445K	216	25	426	24,000	201
HerculesWS	568K	310	0	431	25,000	889
YARP (Robot Platform)	718K	168	31	1,000	42,000	406
uWSGI	102K	377	11	59	6,300	895
DASH C++	91K	58	0	77	6,900	78
Shed Skin	103K	28	0	615	10,000	152

automated inspection mechanisms, the study ensures systematic and consistent code quality assessments across different open-source projects.

C. Manual Validation of Findings

After collecting the analysis output from SonarQube, each reported issue is manually reviewed to distinguish true positives from false positives that may have been incorrectly flagged as vulnerabilities. This manual validation process is essential to ensure the accuracy and reliability of the static analysis results, as automated tools can sometimes produce misleading warnings due to limited context understanding or conservative rule enforcement. During manual review, the code context surrounding each issue is carefully inspected to assess the correctness of control flow and to verify the intended semantic behavior. This includes analyzing logic conditions, checking for appropriate bounds and input validations, and confirming the correct use of secure APIs and best practices. By performing this thorough inspection, it becomes possible to accurately identify actual vulnerabilities and eliminate false alarms that could otherwise mislead project stakeholders or skew the overall assessment. Certain specific cases and representative examples encountered during this review process are further examined and discussed in detail in the case study presented in Section V.

IV. Empirical Study Results

This section presents the quantitative results of our static analysis on selected open-source projects using SonarQube, addressing RQ1: To what extent do actively maintained open-source software (OSS) projects remain exposed to security vulnerabilities despite the use of widely adopted static analysis tools? We report the volume and distribution of detected security vulnerabilities, and assess the precision and coverage of SonarQube's findings based on manual validation.

A. Overall Detection Statistics

Table I provides a summary of vulnerabilities and code quality metrics for a set of open-source projects, as identified by SonarQube. The table includes information on lines of code (LOC) analyzed by SonarQube, number of contributors, and counts for issues related to security, reliability, maintainability, and security hotspots based on software quality. Each issue is defined as below: Security is the protection of the software from unauthorized access, use, or destruction. For example, weak versions of SSL and TLS would be vulnerable to attacks. Reliability is a measure of how your software is capable of maintaining its level of performance under stated conditions for a stated period of time. For example, it refers to cases where you access elements beyond the declared range of an array. Maintainability refers to the ease with which you can repair, improve and understand software code. For example, the datatype is defined incorrectly. Security Hotspot highlights security-sensitive code that the developer needs to review. Hard-coded passwords are considered security hotspots because they can be easily discovered and exploited by attackers. Buffer overflows, for example when using functions like `strcpy` without proper bounds checking, are also critical security hotspots as they can lead to arbitrary code execution or system crashes.

Projects range from 19K LOC (Unity Test) to 2.3M LOC (Zephyr). Larger codebases (e.g., kamailio, Zephyr, YARP) are generally more complex and likely to contain more vulnerabilities and maintenance challenges. The project with the most security issues is cURL (308 issues), followed by Linux Test Project (60) and kamailio (27). Several projects like Edenlive, YARP, HerculesWS, DASH C++, and Shed Skin report 0 security issues. Zephyr has a very large codebase (2.3M LOC) and many contributors (2.8K) but only 24 security issues,

suggesting well-maintained code. Projects with high LOC and many contributors (e.g., Zephyr, kamailio) often have more security and maintainability issues, highlighting challenges in large-scale collaborative development. Smaller projects or those with fewer contributors may still have high maintainability issues (e.g., Shed Skin).

B. Correlation Analysis of Software Quality Metrics

Understanding interactions among software quality attributes is vital for evaluating the security of open-source projects. Analyzing correlations across metrics such as LOC, contributors, and maintainability reveals which factors scale together and which remain independent, highlighting where vulnerabilities are likely to emerge. Figure 1 shows the correlation matrix derived from Table I, offering valuable insights into the relationships among key software quality and security attributes in open-source projects.

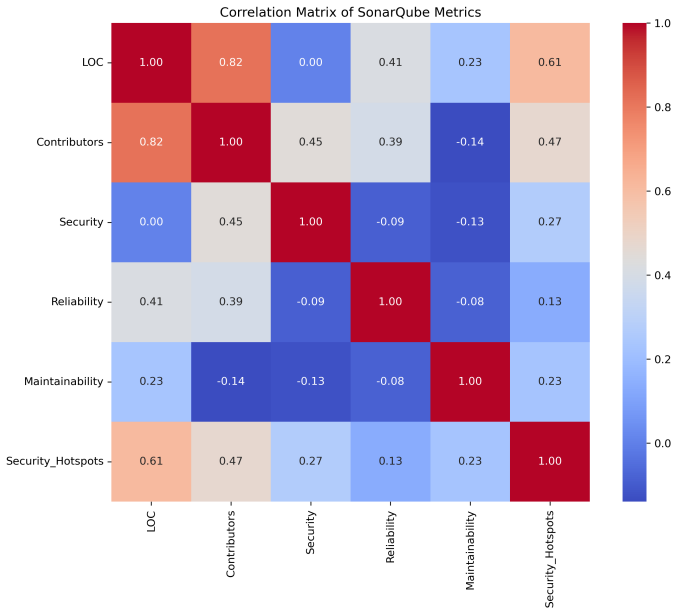

Fig. 1: Correlation Matrix of Software Quality and Security Metrics in Open-Source Projects Evaluated by SonarQube

A strong positive correlation (0.82) was observed between lines of code (LOC) and the number of contributors, indicating that larger codebases tend to require or attract a greater number of contributors to support ongoing development and maintenance efforts. Additionally, LOC exhibited a moderate positive correlation with security hotspots (0.61), suggesting that more extensive projects inherently contain a greater number of potentially vulnerable code segments that necessitate careful security reviews. Notably, the correlation between contributors and security issues was moderate (0.45), implying that an increase in contributor count may be associated with a slight rise in security issues, possibly due to the complexities introduced by diverse coding practices and more frequent code changes. In contrast, reliability issues demonstrated relatively weak correlations with LOC (0.48) and contributors (0.33), highlighting that reliability is influenced more by architectural and design decisions rather than sheer project size or community size alone. Maintainability issues showed only weak correlations with LOC (0.32) and contributors (0.25), underscoring that maintainability challenges are often driven by technical debt, inconsistent design practices, and code complexity rather than directly by project scale or contributor count. Overall, these findings emphasize that while certain metrics such as contributor count and security hotspots scale with project size, other critical quality aspects such as reliability and maintainability require deliberate design choices and continuous engineering attention. This analysis underscores the necessity for robust code review processes, proactive security practices, and consistent architectural discipline to ensure long-term software quality and resilience regardless of project scale.

Contributor numbers vary significantly across projects, and this variation offers important insights into their potential impact on security outcomes. Figure 2 illustrates the relationship between the number of contributors and the number of security issues across various open-source projects. While some projects with a large number of contributors, such as cURL with 1,474 contributors and 308 security issues, exhibit higher vulnerability counts, other large-scale projects like Zephyr show relatively few security issues despite having 2,800 contributors. In contrast, smaller projects generally display fewer security issues, though this trend is not consistently observed across all cases. The computed correlation coefficient of approximately 0.45 suggests a weak to moderate positive correlation, indicating that as the number of contributors increases, the number of security issues may also slightly increase. However, this relationship is neither strong nor strictly linear. This analysis, further supported by the correlation matrix, underscores that a higher number of contributors does not inherently result in fewer security vulnerabilities. On the contrary, projects like cURL demonstrate that even large and active communities can face significant security challenges, especially when managing legacy codebases and complex functionalities.

Another key aspect of our analysis examines the relationship between LOC and the number of detected security vulnerabilities, as shown in Figure 3. This bubble chart visualizes each project based on its codebase size and vulnerability count, with bubble size representing the number of contributors. The result reveals no clear linear correlation between LOC and the number of detected security vulnerabilities, as evidenced by the scattered distribution of projects and a correlation coefficient near zero. While some projects with large codebases, such as cURL, report a high number of vulnerabilities, others with even greater LOC, like Zephyr, show relatively few. This indicates that code size alone is not a reliable predictor of security risk. Instead, factors such as code quality, development discipline, and the limitations of static analysis tools appear to play a

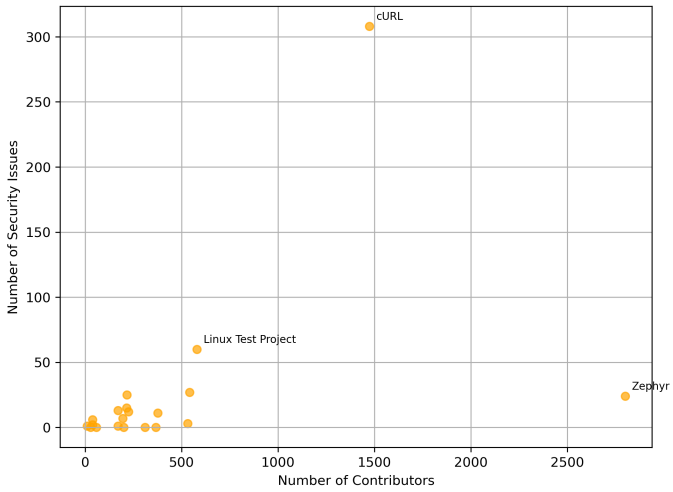

Fig. 2: Relationship Between Number of Contributors and Security Issues (Correlation Coefficient: 0.45)

more significant role. These results highlight the importance of context-aware evaluation strategies that go beyond surface-level code metrics when assessing the security of open-source software.

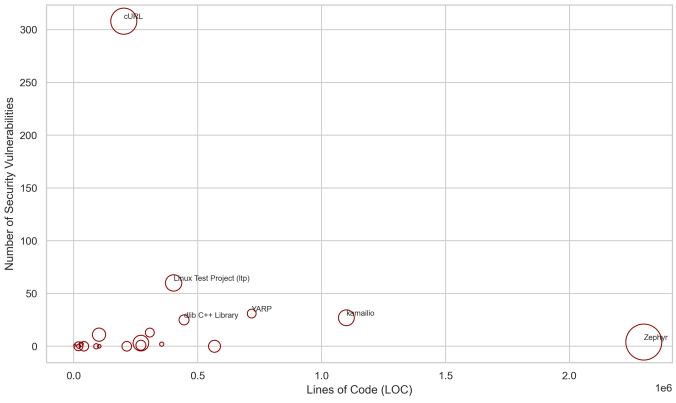

Fig. 3: Relationship Between Lines of Code (LOC) and Security Vulnerabilities (Correlation Coefficient: 0.00). Bubble size represents contributor count.

C. Summary of Findings and Answer to RQ1

The project table and correlation matrix reveal that larger OSS projects with more code and contributors tend to have more security hotspots, likely due to increased complexity and a broader attack surface. However, having more contributors does not always reduce risk and may introduce issues through varied coding styles and frequent changes. Maintainability and reliability appear less related to size, suggesting stronger links to architecture, code consistency, and technical debt. Overall, static analysis tools and active maintenance alone are not enough to ensure security. They must be supported by secure coding practices, code reviews, and proactive vulnerability management. These findings inform the qualitative examples in Section V.

V. Case Studies

To better understand the practical implications of SonarQube's capabilities and limitations, we present three case studies based on real findings from our analysis of open-source projects. These case studies address RQ2: What types of limitations, such as the omission of context-specific vulnerabilities or a high rate of false positives, undermine the practical effectiveness of these tools in supporting secure software development? Each case highlights a different category of behavior: vulnerabilities missed by SonarQube, false alarms generated by overly broad rules, and successful identification and validation of a true positive.

A. Case Study I: `sprintf` Vulnerabilities

SonarQube detected `sprintf` vulnerabilities which can lead to CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) in `SST C API` [28], [29] as well as all 4 vulnerabilities in `LF Reactor-C` [32]. SonarQube also provided suggestions to fix these `sprintf` vulnerabilities by using `snprintf`, which takes an additional argument `size_t n`, the maximum buffer capacity to fill, thereby preventing buffer overflow. Below is an example code snippet of the vulnerability found by SonarQube:

```
char str[6];
sprintf(str, "%u", used_port);
```

We fixed the code above as suggested by SonarQube as shown below:

```
char str[6];
snprintf(str, sizeof(str),
         "%u", used_port);
```

After our fix was merged, SonarQube automatically verified that the vulnerability was removed. We were able to fix all 10 `sprintf` vulnerabilities detected by SonarQube in `SST C API` as well as all 4 vulnerabilities in `LF Reactor-C` [32].

B. Case Study II: Insecure Pseudorandom Number Generator

In `SST C API`, SonarQube detected use of an insecure Pseudorandom Number Generator (PRNG), which can lead to the vulnerabilities caused by non-cryptographic PRNGs (e.g., CVE-2013-6386, CVE-2008-4102). The code with this vulnerability is shown below, using the `rand()` function in the standard library to generate a cryptographic key.

```
<stdlib.h>
int plaintext_buf_length = rand() %
    (MAX_KEY_SIZE + 1 - MIN_KEY_SIZE) +
    MIN_KEY_SIZE;
```

Based on suggestions by SonarQube, we fixed the PRNG using the function provided by the OpenSSL library as shown below.

```
#include <openssl/rand.h>
int secure_rand(int min, int max) {
  unsigned char buf[4];
  if (RAND_bytes(buf, sizeof(buf)) != 1) {
    return -1; }
  unsigned int num =
    ((unsigned int)buf[0] << 24) |
    ((unsigned int)buf[1] << 16) |
    ((unsigned int)buf[2] << 8) |
    ((unsigned int)buf[3]);
  return (num % (max - min + 1)) + min;
}
int plaintext_buf_length =
  secure_rand(MIN_KEY_SIZE, MAX_KEY_SIZE);
```

This fix was automatically detected by SonarQube and the security warning disappeared.

C. Case Study III: Weak Asymmetric Cryptography

SonarQube detected the use of weak asymmetric cryptography configurations in `SST C API`, which can lead to CWE-780, Use of RSA Algorithm without Optimal Asymmetric Encryption Padding (OAEP), as shown below.

```
public_encrypt(buf, buf_len,
  RSA_PKCS1_PADDING,
  (EVP_PKEY *)ctx->pub_key,
  &encrypted_length);
```

SonarQube provided the background and suggested a fix, which we followed to fix the vulnerability as shown below.

```
public_encrypt(buf, buf_len,
  RSA_PKCS1_OAEP_PADDING,
  (EVP_PKEY *)ctx->pub_key,
  &encrypted_length);
```

As soon as the fix was merged, SonarQube automatically detected the change and cleared the vulnerability in the security hotspot report. This change also affected the other ends of asymmetric key encryption and decryption. Thus, this also helped improvements of other parts of the code base written in other languages such as Java and Node.js.

D. Case Study IV: Unsafe use of `strcpy`, `strncpy`, and `strlen`

In `SST C API`, SonarQube flagged the use of `strcpy` and `strncpy` as security hotspots, as the example show below.

```
strncpy(dest, src, dest_size);
```

We applied the best practices suggested by SonarQube to such cases as shown below, by setting the last element of the `char` array to the null character (`0`), and if the copy result is null-terminated.

```
dest[dest_size - 1] = 0;
strncpy(dest, src, dest_size);
if (dest[dest_size - 1] != 0) {
  print_error_exit("Problem found ....\n");
}
```

However, even after we applied the best practices, SonarQube still flagged the code above as a security hotspot. We consider this case a false positive in the SonarQube security analysis. We found similar false positive issues with `strlen`, where even the safe code gets flagged by SonarQube.

Our conjecture on these false positives is that the static analysis of SonarQube is intended to prioritize any suspicious or potential security vulnerabilities over minimizing the false positives. Also, these cases are more subtle and require in-depth analysis of the code, considering the usage context of the functions and surrounding code, including control flows such as `if-else` statements, which will significantly affect the performance and scalability of the code security analysis. We acknowledge that this can be a design decision rather than a limitation or oversight by SonarQube.

E. Summary of Findings and Answer to RQ2

These case studies reveal that while SonarQube performs effectively in flagging certain classes of security issues, its limitations become evident when faced with logic-based flaws or context-sensitive practices, as well as false positives. Specifically, we found 9 false positives on `strlen` and 2 false positives on `strncpy` remaining even after we applied best practice fixes, leading to approximately a false positive rate of 21% (11/52) among the security hotspots discovered by SonarQube. Manual validation and secure coding practices remain essential complements to automated static analysis tools.

VI. Conclusion

Ensuring the security of open-source software remains a significant challenge, especially as many projects operate without dedicated resources for thorough security auditing. This study aimed to analyze both the quantity and nature of security vulnerabilities present in real-world open-source projects, while also examining the practical limitations of static analysis tools in identifying these issues effectively. Our analysis showed that larger open-source projects faced more security vulnerabilities due to complexity, while maintainability and reliability were shaped more by design and code quality. Static analysis tools helped detect clear issues but often missed complex flaws and produced false positives, highlighting the need for manual validation to ensure effective security auditing. Based on the experimental results, our findings underscore the limitations of relying solely on static analysis tools for securing open-source software. While these tools can effectively detect straightforward vulnerabilities, they fall short in identifying complex, context-dependent issues. This highlights the critical need for

combining automated analysis with manual review to achieve more comprehensive and trustworthy security assessments.

This study is limited in scope to a specific set of open-source projects and a single static analysis tool. While manual validation was performed to improve accuracy, it may still be subject to evaluator bias or oversight. Future work can broaden analysis by including more diverse open-source projects and evaluating additional static analysis tools. This would improve generalizability, reveal tool-specific strengths and weaknesses, and enhance automated security auditing in open-source development.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation (NSF) under Award No. OIA-1946391, OIA-2445877, and POSE-2449200 (An Open-Source Ecosystem to Coordinate Integration of Cyber-Physical Systems).

REFERENCES

[1] Sean P Goggins, Matt Germonprez, and Kevin Lumbard. Making open source project health transparent. *Computer*, 54(8):104–111, 2021.

[2] Shao-Fang Wen. Software security in open source development: A systematic literature review. In *2017 21st Conference of Open Innovations Association (FRUCT)*, pages 364–373. IEEE, 2017.

[3] Ruyan Lin, Yulong Fu, Wei Yi, Jincheng Yang, Jin Cao, Zhiqiang Dong, Fei Xie, and Hui Li. Vulnerabilities and security patches detection in oss: a survey. *ACM Computing Surveys*, 57(1):1–37, 2024.

[4] SonarSource. SonarQube Static Code Analysis. https://www.sonarsource.com/products/sonarqube/. Accessed: July 28, 2025.

[5] Roman Ivanov, Richard Veach, et al. Checkstyle 10.26.1 – checkstyle. https://checkstyle.org/. Accessed: July 28, 2025.

[6] David Hovemeyer, Bill Pugh, et al. FindBugs™ - Find Bugs in Java Programs. https://findbugs.sourceforge.net/. Accessed: July 28, 2025.

[7] Jones Yeboah and Saheed Popoola. Efficacy of static analysis tools for software defect detection on open-source projects. In *2023 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1588–1593. IEEE, 2023.

[8] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pages 544–555, 2022.

[9] Giammaria Giordano, Fabio Palomba, and Filomena Ferrucci. A preliminary conceptualization and analysis on automated static analysis tools for vulnerability detection in Android apps. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 201–208. IEEE, 2022.

[10] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. Some SonarQube issues have a significant but small effect on faults and changes. a large-scale empirical study. *Journal of Systems and Software*, 170:110750, 2020.

[11] OpenHub. OpenHub: Discover Open Source Projects. https://openhub.net/. Accessed: July 28, 2025.

[12] Black Duck. Open source software vulnerabilities found in 86% of codebases. Security Magazine, February 2025.

[13] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–420. IEEE, 2015.

[14] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th international conference on Software engineering*, pages 356–366, 2014.

[15] Felix Fischer, Jonas Höbenreich, and Jens Grossklags. The effectiveness of security interventions on GitHub. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2426–2440, 2023.

[16] Tom Mens et al. Mitigating security issues in GitHub Actions. In *2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM 2nd International Workshop on Software Vulnerability*. ACM/IEEE, 2024.

[17] Jessy Ayala and Joshua Garcia. An empirical study on workflows and security policies in popular GitHub repositories. In *2023 IEEE/ACM 1st International Workshop on Software Vulnerability (SVM)*, pages 6–9. IEEE, 2023.

[18] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25(2):1419–1457, 2020.

[19] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. Challenges for static analysis of Java reflection-literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 507–518. IEEE, 2017.

[20] Synopsys. Coverity Scan Static Analysis Service. https://scan.coverity.com/, 2023. Accessed: July 16, 2025.

[21] GitHub. CodeQL: Semantic Code Analysis Engine. https://codeql.github.com/, 2023. Accessed: July 16, 2025.

[22] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[23] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[24] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

[25] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481. IEEE, 2016.

[26] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[27] Hokeun Kim, Eunsuk Kang, Edward A Lee, and David Broman. A toolkit for construction of authorization service infrastructure for the Internet of Things. In *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*, pages 147–158, 2017.

[28] Dongha Kim, Yeongbin Jo, Taekyung Kim, and Hokeun Kim. SST v1. 0.0 with C API: Pluggable security solution for the Internet of Things. *SoftwareX*, 22:101390, 2023.

[29] Dongha Kim, Yeongbin Jo, Hokeun Kim, Carlos Beltran Quinonez, Taekyung Kim, and Jose Felix. Secure Swarm Toolkit (SST) C API Repository. https://github.com/iotauth/sst-c-api.

[30] Christian Menard, Marten Lohstroh, Soroush Bateni, Matthew Chorlian, Arthur Deng, Peter Donovan, Clément Fournier, Shaokai Lin, Felix Suchert, Tassilo Tanneberger, Hokeun Kim, Jeronimo Castrillon, and Edward A. Lee. High-performance deterministic concurrency using Lingua Franca. *ACM Transactions on Architecture and Code Optimization*, 20(4):1–29, 2023.

[31] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A Lee. Toward a Lingua Franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(4):1–27, 2021.

[32] Soroush Bateni et al. Lingua Franca Reactor-C Repository. https://github.com/lf-lang/reactor-c.

[33] TIOBE Software. TIOBE Index for July 2025. https://www.tiobe.com/tiobe-index/. Accessed: July 10, 2025.

[34] Lan Zhang, Qingtian Zou, Anoop Singhal, Xiaoyan Sun, and Peng Liu. Evaluating large language models for real-world vulnerability repair in c/c++ code. In *Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics*, pages 49–58, 2024.