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

Donghoon Kim, Arkansas State University
Hokeun Kim, Arizona State University

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

ARKANSAS STATE UNIVERSITY

ARIZONA STATE UNIVERSITY

Motivation

- Open-source software (OSS) is the backbone of modern software infrastructure,
  - E.g., operating systems, web frameworks, cryptographic libraries, and data platforms.
- Its collaborative and transparent development model accelerates innovation, but small flaws can propagate through dependency chains and impact many downstream systems.
- Automated vulnerability detection has grown rapidly, and static analysis tools are widely used
  - However, coverage gaps and false alarms still persist.
- This research provides empirical evidence from real projects and analyzes these tools' strengths and limitations.

Goals & Research Question (RQ)

- Measure security exposure in actively maintained OSS
  - (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?
- Identify strengths and limits of static analysis
  - (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?
- Validate fixes and observe tool behavior
  - Apply remediations and re-scan to see how the tool responds

Contributions of This Work

- This work presents an empirical assessment of security exposure in open-source software.
  - 20 real projects using a mixed quantitative and qualitative analysis.
- It characterizes static analysis tools and clearly explains their limitations.
  - Case studies illustrate false positives and context-dependent vulnerabilities.

Methodology: Project Section

- Open-source projects from the OpenHub platform to ensure the robustness and generalizability of our analysis
- Selected 20 projects based on the following criteria:
  - Programming languages: C/C++ focus; include Java/Python for breadth
  - Activity: recent commits; multiple contributors
  - LOC (Lines of Code): at least 10,000 LOC
    - Excluded large projects to prevent analysis failures or impractical analysis processing time

Static Analysis Tool: SonarQube

- We use SonarQube cloud, a cloud-based service
- SonarQube is configured to import project from GitHub repositories
  - Target repository is forked into the users's personal repository
  - 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 conduct a comprehensive evaluation.
- Manual validation process is necessary to identify actual vulnerabilities and eliminate false positive

Experimental Study Results

- Table 1 is a summary of vulnerabilities based on code quality metrics identified by SonarQube.

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

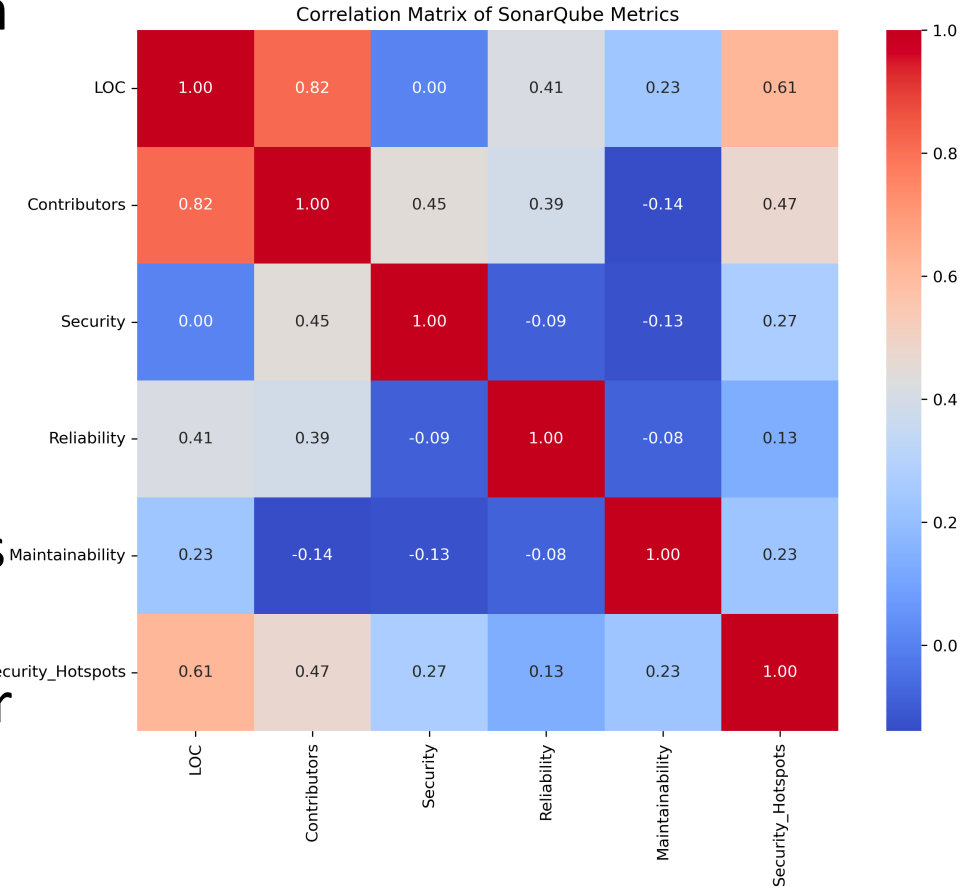
Project	LOC	Contributors	Security	Reliability	Maintainability	Security Hotspots
Kamailio	1.1M	542	27	190	29,000	3,400
cURL	202K	1,474	308	45	4,600	1,100
Linux Test Project (LTP)	403K	579	60	709	15,000	2,100
Neat Project	30K	39	2	73	1,200	84
PROJ (Cartographic Library)	271K	215	1	334	15,000	275
Swift 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

Code Quality Metrics in SonarQube

- **Security** is the protection of the software from unauthorized access, use, or destruction.
  - For example, weak versions of SSL and TLS
- **Reliability** is a measure of how your software is capable of maintaining its level of performance
  - For example, accessing 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.
  - For example, Hard-coded passwords and buffer overflow (e.g., strcpy)

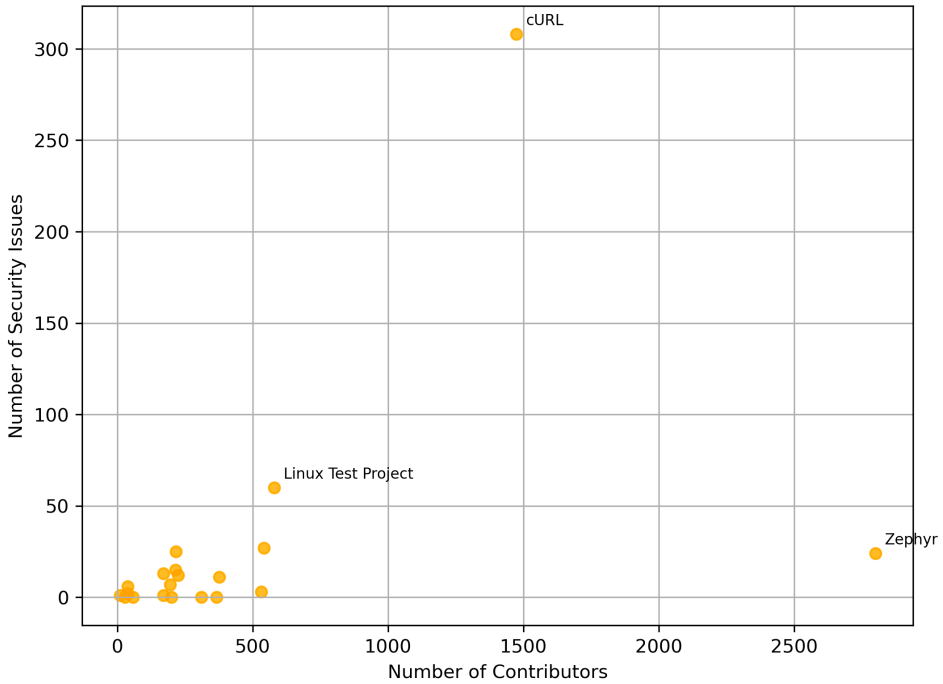
Correlation Analysis of Software Quality Metrics

- A strong positive correlation (0.82) between LOC and the number of contribution
- A moderate positive correlation (0.61) between LOC and security hotspots
- A moderate correlation (0.45) between contributors and security issues
  - An increase in contributor count may be associated with a slight rise in security issues



Contributors and Security Issues

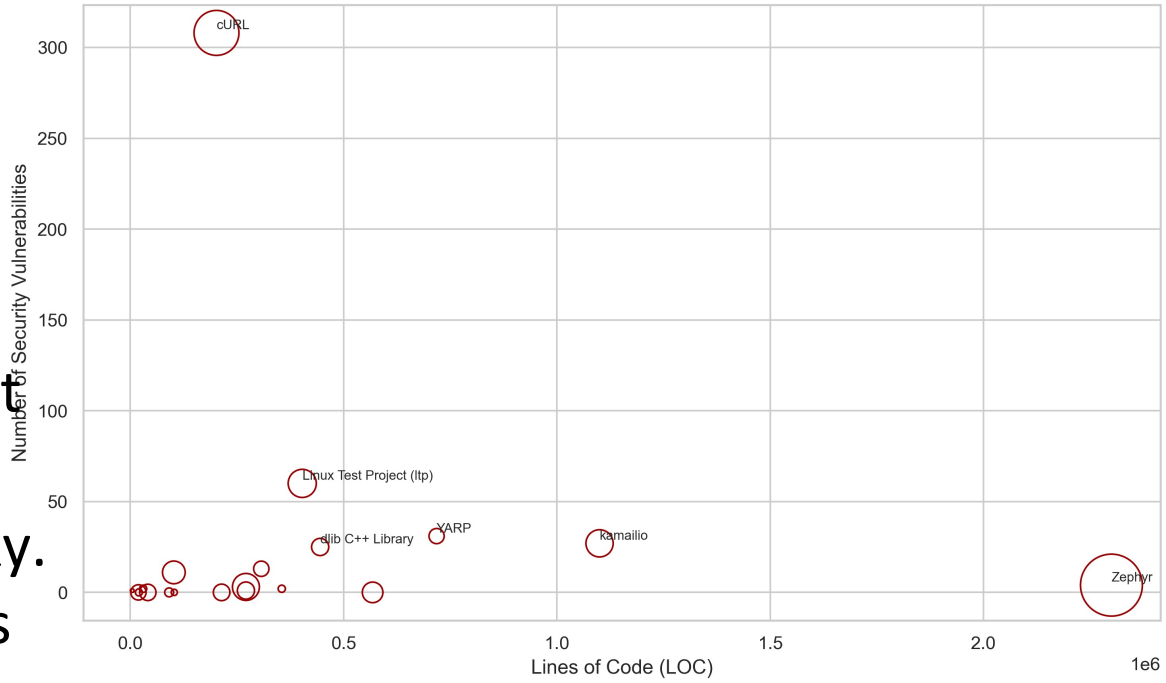
- Correlation coefficient is 0.45 (weak to moderate positive)
- Implication: having more contributors does not guarantee safer code;
  - Process and guardrails matter more than headcount.



- Example contrast:
  - ❑ A highly popular project (cURL) can have many issues
  - ❑ Even more contributors (Zephyr) can have few, showing the relationship isn't linear.

LOC vs Security Vulnerabilities

- Show almost no correlation (~0.00): project size does not predict vulnerability count
- Implication: big project can be clean; small project can still be risky.
- Security vulnerabilities are more related to engineering practices (safe APIs, reviews, CI gates), not size



Bubble size represents contributor count.

Summary of Findings and Answer to RQ1

- (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?
- Bigger projects show more security hotspots;
- More contributors do not guarantee safety;
- Maintainability/reliability are only weakly related to size.
- Answer to RQ1: Actively maintained OSS still has non-trivial vulnerabilities
- In addition, static analysis alone is not enough; it must be paired with secure coding and code reviews.

Case Studies

- Purpose: To clarify the practical implications of static analysis (SonarQube) capabilities and limitations.
- This work presents four case studies based on real findings from analyses of open-source projects.
- For precise remediation and verification via re-scan, the case studies use open-source projects that we can modify:
  - Secure Swam Toolkit (SST) C API
  - Lingua Franca (LF) Reactor-C.

Case Study 1: Unsafe C string API

- Problem: Static analysis flagged uses of “sprintf” / similar unsafe string function; buffer overflow risk (CWE-119)

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

- Fix: This work replaced them with bounded alternatives (e.g., snprintf) and used helper wrappers to enforce length checks.

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

- Scope: 10 instances fixed in SST C API and 4 in LF Reactor-C
- Outcome: A re-scan removed the findings, confirming the fix worked.

Case Study 2 & 3: Crypto Fixes

- Case Study 2: Weak Randomness (PRNG; Pseudo Random Number Generator)
  - Problem: rand() used for security-sensitive values
    - Predictable, not CSPRNG (Cryptographically Secure PRNG)
  - Fix: switch to OS/library CSPRNG (e.g., OpenSSL RAND_bytes)
  - Outcome: warning cleared on re-scan; security posture improved
  - Lesson: policy—no rand() in security paths; provide a safe RNG helper
- Case Study 3: Weak Asymmetric Cryptography (RSA)
  - Problem: legacy padding (e.g., RSA_PKCS1_PADDING) → weaker security defaults
  - Fix: adopt RSA_PKCS1_OAEP_PADDING; centralize via crypto wrapper with safe defaults
  - Outcome: hotspot removed; interoperability preserved (downstream clients)
  - Lesson: “secure-by-default” wrappers; CI gate to block unsafe crypto settings

Case Study 4: Typical False Positive

- Unsafe use of strcpy, strncpy, and strlen
  - Buffer overflow
- In SST C API, SonarQube flagged the use of strcpy and strncpy

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

- Best practices by adding the last element of the char array to the null character (0)

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

- SonarQube continued to mark the snippet as a security hotspot
  - Insufficient handling of context such as defensive sentinel checks.

Summary of Findings and Answer to RQ2

- Static analysis is effective at catching straightforward issues (e.g., unsafe C string APIs, weak randomness, legacy RSA settings),
- However, it struggles with context-dependent and produces notable false positives (e.g., strncpy/strlen).
- 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?
- Answer to RQ2:
  - Static analysis alone is insufficient for practical assurance.
  - It should be paired with secure-by-default APIs, and human code review to translate findings into reliable fixes.

Conclusion

- This work shows that static analysis tools reliably catch clear, pattern-based issues but often miss vulnerabilities that depend on code context.
- Larger codebases and teams tend to create more review hotspots, yet size and headcount do not guarantee safer software.
- Human code review and rule tuning are necessary to reduce false positives and maintain developer trust.
- Future work will expand the set of projects and tools, perform cross-validation, and strengthen practical guardrails in typical workflows.