Finding 709 Defects in 258 Projects: An Experience Report on Applying CodeQL to Open-Source Embedded Software (Experience Paper)

MINGJIE SHEN, Purdue University, USA
AKUL ABHILASH PILLAI, Purdue University, USA
BRIAN A. YUAN*, Purdue University, USA
JAMES C. DAVIS, Purdue University, USA
ARAVIND MACHIRY, Purdue University, USA

Embedded software is deployed in billions of devices worldwide, including in safety-sensitive systems like medical devices and autonomous vehicles. Defects in embedded software can have severe consequences. Many embedded software products incorporate Open-Source Embedded Software (EMBOSS), so it is important for EMBOSS engineers to use appropriate mechanisms to avoid defects. One of the common security practices is to use Static Application Security Testing (SAST) tools, which help identify commonly occurring vulnerabilities. Existing research related to SAST tools focuses mainly on regular (or non-embedded) software. There is a lack of knowledge about the use of SAST tools in embedded software. Furthermore, embedded software greatly differs from regular software in terms of semantics, software organization, coding practices, and build setup. All of these factors influence SAST tools and could potentially affect their usage.

In this experience paper, we report on a large-scale empirical study of SAST in EMBOSS repositories. We collected a corpus of 258 of the most popular EMBOSS projects, and then measured their *use* of SAST tools via program analysis and a survey (N=25) of their developers. Advanced SAST tools are rarely used – only 3% of projects go beyond trivial compiler analyses. Developers cited the perception of ineffectiveness and false positives as reasons for limited adoption. Motivated by this deficit, we applied the state-of-the-art (SOTA) CodeQL SAST tool and measured its ease of use and actual effectiveness. Across the 258 projects, CodeQL reported 709 true defects with a false positive rate of 34%. There were 535 (75%) likely security vulnerabilities, including in major projects maintained by Microsoft, Amazon, and the Apache Foundation. EMBOSS engineers have confirmed 376 (53%) of these defects, mainly by accepting our pull requests. Two CVEs were issued. Based on these results, we proposed pull requests to include our workflows as part of EMBOSS Continuous Integration (CI) pipelines, 37 (71% of active repositories) of these are already merged. In summary, we urge EMBOSS engineers to adopt the current generation of SAST tools, which offer low false positive rates and are effective at finding security-relevant defects.

CCS Concepts: • **Computer systems organization** → **Embedded software**; • **Security and privacy** → **Software and application security**; **Systems security**.

Additional Key Words and Phrases: Empirical Software Engineering, Static Application Security Testing (SAST)

*Work performed while at Purdue University.

Authors' Contact Information: Mingjie Shen, Purdue University, West Lafayette, USA, shen497@purdue.edu; Akul Abhilash Pillai, Purdue University, West Lafayette, USA, pillai23@purdue.edu; Brian A. Yuan, Purdue University, West Lafayette, USA, bryuan@student.ethz.ch; James C. Davis, Purdue University, West Lafayette, USA, davisjam@purdue.edu; Aravind Machiry, Purdue University, West Lafayette, USA, amachiry@purdue.edu.

ACM 2994-970X/2025/7-ARTISSTA048
https://doi.org/10.1145/3728923

ACM Reference Format:
Mingjie Shen, Akul Abhilash Pillai, Brian A. Yuan, James C. Davis, and Aravind Machiry. 2025. Finding 709 Defects in 258 Projects: An Experience Report on Applying CodeQL to Open-Source Embedded Software (Experience Paper). *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA048 (July 2025), 24 pages. https://doi.org/10.1145/3728923

1 Introduction

Societies rely on embedded systems and IoT devices in our transportation [5], traffic management [92], resource distribution [78, 83], homes [8], and in many other ways [4]. The Embedded Software (EmS) that enables these devices must be free of vulnerabilities. Such vulnerabilities have far-reaching consequences [15, 23, 70, 76, 105] due to the pervasive and interconnected nature of embedded devices. Additionally, Open-Source Software (OSS) plays an important role in EmS development [10, 41, 63]. For instance, FreeRTOS [19] and Zephyr [94], two of the most popular and industry-endorsed Real Time Operating Systems (RTOSes), are open-source. Previous studies [12, 87] show that Open-Source Embedded Software (EMBOSS) are riddled with security vulnerabilities, specifically memory safety issues.

Several static and dynamic analysis-based tools exist for vulnerability detection. Dynamic analyses, such as fuzzing [67], are known to be effective at precise vulnerability detection. However, applying these techniques to embedded systems is challenging [74, 102] because of their close interaction with hardware and its diversity. Static analysis techniques, specifically SAST tools, are best suited as they do not need to execute the embedded software or EMBOSS. On the other hand, most existing works [53, 77, 82, 85] on evaluating SAST tools focus on traditional (*i.e.,* non-embedded) software. However, embedded software differs from traditional software in organization, architecture, build system, and toolchains [102]. *What would be the effectiveness of SAST tools on EMBOSS?*

In this experience paper, we present the first empirical study on the use of SAST tools to detect security vulnerabilities in EMBOSS. For our study, we curated a corpus of 258 popular EMBOSS projects from GitHub. We used this corpus for the three phases of our investigation.

(1) Measuring the use of SAST in EMBOSS: First, we combined automated analysis of CI workflows from the corpus and a survey of the project developers to understand the prevalence of SAST usage. We found that only 10 (4%) projects use explicit SAST tools as part of their CI workflows. Developers of the remaining projects are aware of SAST tools but do not use them on EMBOSS projects as they believe that the effectiveness of SAST tools on their repositories is low. It is unclear whether this belief is accurate.

(2) Selecting and Configuring a SAST Tool: Next, to fill this knowledge gap, we aim to understand the effectiveness of SAST on EMBOSS. We conducted a preliminary analysis and found that the CodeQL was the most effective available SAST tool. First, the default setup of CodeQL (that works well for traditional software) failed on EMBOSS repositories. Furthermore, the default analysis resulted in a lot of false positives. We tackled this by manually (with minimal engineering effort) creating CI workflows enabling the execution of CodeQL on EMBOSS repositories. Second, the default analyses queries of CodeQL resulted in a lot of false positives. We tackled this by filtering out certain queries and modifying relevant queries.

(3) Measuring the effectiveness of SAST in EMBOSS: We executed our CI workflows with modified CodeQL queries and found a total of 709 defects, with 535 (75%) being security vulnerabilities. On a per-report basis, CodeQL exhibits a false positive rate of 34%, but this is due to a few outlier rules and projects. For most studied repositories, the false positive rates were low. We reported 586 of defects we found. Developers have already confirmed 376 (53%) of these defects, mainly by accepting our patches. We also raised pull requests to 129 EMBOSS, integrating our manually created workflows (enabling running CodeQL) into their CI pipeline, out of which 37 (71% (Active) and 29% (Total)) are already accepted. We hope that our findings: (1) provide evidence of the effectiveness of SAST tool on EMBOSS repositories; (2) encourage EMBOSS developers to adopt SAST tools; and (3) motivate researchers to work on techniques to automatically integrate SAST tools in CI workflows.

In summary, this experience report contributes:

- **(Empirical Study)** We presented the first study on the prevalence, challenges, and effectiveness of using SAST tools in EMBOSS, via automated and manual analysis and a developer study.
- **(Lessons Learned)** We summarize our experience in four lessons learned (§6), capturing our experiences using a SOTA SAST tool, finding hundreds of defects, reporting them, and integrating the SAST tool in the CI pipeline of EMBOSS repositories.
- **(Dataset)** We curated and categorized a list of 258 major EMBOSS projects, accompanied by GitHub workflows for compilation to permit the execution of static and dynamic analysis tools. This is the first large-scale embedded software dataset with compilation infrastructure.
- **(Impact)** Using off-the-shelf CodeQL queries on these workflows, we identified a total of 709 defects (535 (75%) security vulnerabilities) across this dataset, including projects maintained by the Apache Foundation, Microsoft, and Amazon. We reported 586 of these defects, of which developers confirmed 376 (53%) of them. We also raised pull requests to 129 projects to integrate CodeQL workflows in their CI pipelines, of which 37 are accepted.

Significance for software engineering: Empirical software security research has a substantial body of knowledge on open-source software, but has focused on IT or general-purpose software. We report on a large-scale experience of applying static analysis to open-source embedded software. Across 258 EMBOSS repositories, the CodeQL SAST tool finds hundreds of defects with low false positive rates in the majority of repositories. Motivated by this knowledge, we recommend that EMBOSS software developers use this tool to easily improve software quality.

2 Background

Here we define Open-Source Embedded Software (EMBOSS) and Dynamic and Static Application Security Testing (SAST).

2.1 Open-Source Embedded Software (EMBOSS)

2.1.1 Definition of Embedded Software and EMBOSS. Embedded software is designed to run on embedded systems, ranging from industrial controllers [22] to IoT devices with resource-constrained microcontrollers [8].

Open-Source Software (OSS) is an essential part of the software supply chain of embedded software applications. A considerable proportion of software products incorporate open-source software in order to reduce costs and develop more competitive products [10]. Application developers re-use many kinds of EMBOSS, but a particularly common dependency is on specialized Real Time Operating Systems (RTOSes) designed for reduced-resource environments (*e.g.,* real-time scheduling, low power consumption, low memory overhead). According to `osrtos.com`, there are 31 different RTOSes [1], with the majority (26) of them being open-source. Examples of RTOSes include RIOT, Contiki, FreeRTOS, and Azure RTOS.

2.1.2 Measuring Project Importance. A common way to measure the importance of an open-source project is the Open Source Security Foundation (OSSF) criticality score [17]. This score is used by security analysts to triage security vulnerabilities when studying a large number of projects [35, 64]. A project's importance is a number between 0 and 1 based on attributes including its popularity, dependents, and level of activity. Ranges correspond to qualitative labels: 0.0-0.2 is considered low criticality, 0.2-0.4 is medium, 0.4-0.6 is high, 0.6-0.9 is critical, and above 0.9 is extremely critical. For examples, the RTOS contiki-os has a criticality score of 0.51 (high), the RTOS Zephyr's score is 0.81 (critical), and the Node.js runtime's score is 0.99 (extremely critical).

2.2 Static Application Security Testing (SAST)

2.2.1 SAST vs. DAST in Embedded Software. In software security analysis, both static (SAST) and dynamic (DAST) application security testing are necessary.

In the context of embedded systems, dynamic analysis (*e.g.,* fuzzing) is more costly and sometimes infeasible when compared to static analysis. Embedded software is coupled to hardware [74], *e.g.,* using hardware-specific interfaces and custom instruction sets. Executing it on custom hardware needs an emulator (support may be lacking [40]) or physical boards (resulting in unscalable testing). Static Application Security Testing (SAST) tools do not require execution, making them attractive to use on embedded software.

2.2.2 Landscape of SAST Tools. There are many open-source and commercial SAST tools. The open-source tools vary in the underlying techniques and corresponding guarantees. There are high-assurance tools, such as IKOS [25], that use abstract interpretation and provide soundness guarantees. However, these tools must be properly configured with suitable abstract domains to avoid false positives – a cumbersome process requiring a formal background. On the other hand, there are best-effort pattern-based tools, such as `cppcheck` [71] and `flawfinder` [101], which can be readily used but do not provide any guarantees. Several works [29, 44, 62, 73] evaluate these tools on non-embedded software and show that they vary in precision, recall, and usability. There are also many commercial SAST tools. Coverity is considered SOTA and allows developers to customize the tool to reduce false positives [52], but its license forbids evaluation in research papers. Other notable tools include Fortify [79], Checkmarx [30], and Veracode [97].

CodeQL is a SOTA [60] open-source SAST tool. CodeQL was released in 2016 by GitHub and is maintained by Microsoft. CodeQL represents code as a relational database and uses relational queries to find defects in the given codebase. It has several static analysis capabilities, such as control flow analysis, data flow analysis, and taint tracking to detect security issues [18]. Furthermore, CodeQL has built-in queries for common security issues (*i.e.,* Common Weakness Enumerations (CWEs)). Security analysts and developers have used CodeQL to find thousands of security vulnerabilities in large and well-tested codebases including the Linux kernel [34, 45, 75]. Since CodeQL is free to use on open-source codebases and its queries are open-source, it is a popular SAST tool within the open-source community.

2.2.3 How SAST is Applied in Modern OSS. Continuous Integration (CI) pipelines [51] have become ubiquitous in the modern software development lifecycle. They automate various software development processes, such as building, testing, and deploying code. By this means, software development has shifted towards the continuous (or near-continuous [14]) integration of changes, allowing deployment at more rapid intervals [42]. SAST and DAST tools are often applied as part of a CI pipeline [21, 68, 69], reflecting the "shift left" trend to assess security throughout the engineering process rather than at fixed intervals.

On GitHub, the main open-source software platform, there are several options for CI frameworks [37], *e.g.,* TravisCI [31], CircleCI [32], and GitHub Actions [28]. The most popular is GitHub Actions because of its close integration with GitHub's platform [49]. The GitHub CI is structured as a set of *workflows* associated with events. Each workflow is comprised of one or more *Actions.* Our Extended Report [91] provides more detail about GitHub workflows.

3 Motivation

Many works [53, 77, 82, 85] emphasize the importance of using SAST tools on software projects, especially in unsafe languages such as C/C++ (which most EMBOSS repositories use). Cybersecurity and government organizations [27, 96] also recommend the use of SAST. But no study measures the prevalence or benefits of SAST in EMBOSS.

Existing studies [16, 29, 44, 62, 73] evaluate the effectiveness of SAST tools on non-embedded software. Embedded software differs from traditional software in organization, architecture, build system, and toolchains [102]. EMBOSS follows a layered organization where each layer exposes fixed functionalities to the one above and relies on those below [89]. To enhance flexibility, inter-layer communication uses function pointers, leading to indirect control flow transfers – a common cause of static analysis imprecision. Additionally, EMBOSS employs an event-driven architecture with handlers triggered by specific events (*e.g.,* interrupts) [84]. These handlers communicate via global objects, creating asynchronous control flows with global pointer manipulations, which challenge flow-based static analysis. Furthermore, EMBOSS relies on diverse, non-standard build systems [89] and exotic compilers (*e.g.,* `avr-gcc`), posing engineering difficulty in applying compilation-based SAST tools. Therefore, it is unclear how challenging it is to use existing SAST tools and how effective they are on EMBOSS.

4 Prevalence Study

Given the potential benefits of SAST tools, we first study the prevalence of their usage in EMBOSS. We curate a *corpus* of major EMBOSS from GitHub (§4.1) and other well-known sources.

4.1 Corpus of Major EMBOSS Projects

Table 1. Summary of repositories in our EMBOSS dataset, grouped by project categories. SLOC calculated with `cloc` [36]. Criticality with the OSSF tool [17]. Data is as of July, 2023. The *Total* row gives medians across corpus, not by category. Medians are rounded to the nearest integer.

Category	**# Repos**	**Example Repo**	**Median GH stars**	**Median SLOC**	**Median Crit. Score**
Hardware Access Library (HAL)	18	grbl	304	98,502	0.44
Device Drivers (DD)	10	TinyUSB	452	20,078	0.41
Network (NET)	54	contik-ng	314	36,345	0.46
Database Access Libraries (DAL)	8	tiny SQL	659	26,977	0.39
File Systems (FS)	5	littlefs	401	11,195	0.49
Parsing Utilities (PAR)	10	mjson	314	2,547	0.41
Language Support (LS)	33	micropython	479	33,389	0.42
UI Utilities (UI)	14	flutterpi	584	56,712	0.46
Embedded Applications (APP)	32	Infinitime	508	22,662	0.39
Operating Systems (OS)	42	FreeRTOS	728	409,668	0.47
Memory Management Library (MML)	4	tinyobjloader-c	242	6,206	0.34
Other General Purpose Library for Embedded Use (GPL)	22	tinyprintf	391	12,742	0.35
Other (OT)	6		368	94,805	0.43
Total	258		406	33,545	0.43

We aim to collect a set of representative and well-engineered EMBOSS. We combine two approaches (Figure 1). First, we searched GitHub for embedded software projects (§4.1.1). Second, we used an external index of RTOSes (§4.1.2).

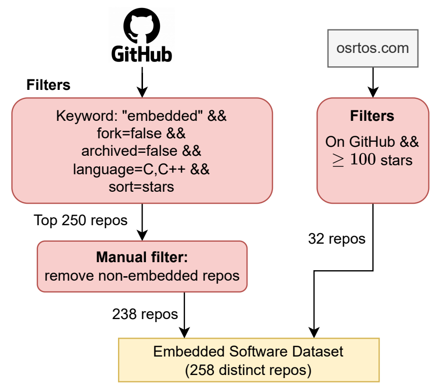


Fig. 1. Two-pronged approach to collecting embedded software dataset. The GitHub search (left side) was performed on April 8, 2023. The osrtos.com search (right side) was performed on June 7, 2023.

4.1.1 EMBOSS from GitHub Search. We searched GitHub for popular embedded software on GitHub. Specifically, we collected original (*i.e.,* non-forked), active (*i.e.,* non-archived) C/C++ embedded software. Figure 1 shows the exact filters for our search. The initial query yields ~20K projects. We sorted them by popularity (operationalized as the number of stars [24]) and collected the top 250. We manually filtered out 12 false positives (non-embedded repositories) based on their READMEs. For instance, we filtered out a machine learning project that had the word "embedded" in its keywords.

4.1.2 EMBOSS from Index of RTOSes. Embedded systems are usually powered by an RTOS, which provides the necessary library and scheduling support for various application components. We collected RTOSes from osrtos.com [1], which maintains a curated list of open-source RTOSes. Specifically, we selected those available on GitHub with >100 stars. This resulted in a total of 32 repositories.

4.2 Analysis of Corpus

We combined the repositories and de-duplicated them, resulting in a total of 258 unique EMBOSS repositories. Table 1 summarizes all projects along with their fine-grained categorization (performed manually). Most repositories are reasonably large, with a median of 33K Source Lines of Code (SLOC) and a maximum >400K SLOC. This is similar to the project sizes examined in other studies [89].

We also measured the repositories using the OSSF criticality measure (§2.1.2). All 13 categories have a median project with "medium" or "high" criticality score; the overall median criticality score is 0.43 (high). This indicates that our corpus includes important projects.

4.3 Study Methodology

We examine the state of practice usage of SAST tools from two views: the use of SAST in the CI workflows of the corpus, and a survey of the project developers in the corpus.

4.3.1 Workflow Analysis. We noticed that 42% (109/258) of the EMBOSS repositories use GitHub workflows to build and test the underlying codebase. We automatically analyzed these workflows to detect the use of SAST tools. Specifically, for each Action used in a workflow, we check if it is a SAST tool by checking its category in the GitHub CI Actions marketplace. We define SAST tools as those whose marketplace category is "code quality" or "security." Next, we manually check every matching Action to validate that it is indeed a SAST tool.

For workflows for which no SAST was found (248/258 of projects), we estimated whether or not this occurred due to errors in our automated analysis, or because they indeed used no SAST. We performed a random sampling of 20 workflows and manually checked them.

To make the measured rate of SAST usage interpretable, we performed the same measurement on the top 5,000 OSS projects deemed "critical" and "extremely critical" according to the OpenSSF criticality score. These projects do not target embedded contexts – none of the projects from our corpus appear in this list.

Results. We found that only 10 (4%) of the repositories use a sophisticated SAST tool. All of these use free SAST tools, specifically, CodeQL. None of them use commercial SAST tools. Of the 10 repositories that use CodeQL, 7 use an out-of-date version.

By comparison to the top 5,000 OSS projects by criticality (without the embedded constraint), we can see how small this adoption rate is. Of the top 5,000 OSS projects we examined for comparison, 958 (19%) use ≥1 SAST tools by our definition.

In our random sampling to check for false negatives in the EMBOSS measure (a random sample of 20 projects), we found only two false negatives, *i.e.,* 10% false negative rate. Both were due to a level of indirection around the use of a SAST tool. `RIOT-OS/RIOT` runs its static tests in a Docker container, and `InfiniTimeOrg/InfiniTime` runs `clang-tidy` in a script.

4.3.2 Developer Survey. To complement our previous workflow analysis, we conducted a developer survey under the supervision of our institution's Institutional Review Board (IRB). The survey aimed to gather insights from projects' maintainers about their security practices and to identify any alternative ways in which they might use SAST tools. Our population of interest was the maintainers of the 248 (96%) of projects that do not use any SAST Actions. For each of these projects, we collected emails of users who recently contributed and emailed them the link to our survey. We were able to find the maintainers' email for 104 (out of 248) projects. There were 15 questions in the survey with an anticipated time of 5 min. The full survey is in Our Extended Report [91].

Results. We got 25 responses (24% response rate), representing 20 distinct repositories. This response rate is comparable to that reported by other works that survey developers from GitHub (*e.g.,* [2, 55]). While the survey's sample size is relatively small, it still offers valuable insights into developers' perspectives, and complements our quantitative measurement results for this research question.

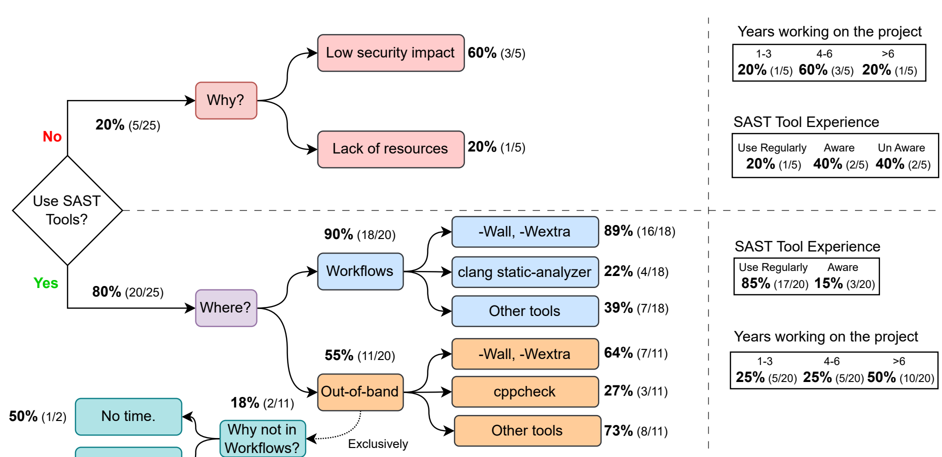


Fig. 2. Summary of our developer survey on the use of SAST tools.

Figure 2 illustrates the responses to our survey. We discuss why developers do and do not use SAST tools.

Use of SAST Tools. Most of the surveyed developers (80% (20/25)) claimed to be using SAST tools. *However, the most commonly used approach was enabling compiler warnings such as* `gcc -Wall -Wextra`*, and developers considered those to be adequate SAST.* Compiler warnings are not effective as they mainly catch simple issues and have high false negative rates. As a simple demonstration of the weaknesses of compiler warnings as SAST, we executed `gcc`'s analyses on a set of test cases from the CodeQL repository. These are simple test cases (<10 lines), each demonstrating a security issue, *e.g.,* using `%s` in `scanf` or passing invalid pointer types to a function call. We compiled these test cases using a recent version of `gcc` (11.4.0) with strict warnings. This configuration of `gcc` found issues in only 17 (21%) defect types. Some simple security issues were flagged, such as the use of `strcpy` instead of `strncpy`. However, more complex ones were missed, such as inconsistent `NULL` checks and use-after-free errors. We provide more details in Our Extended Report [91]. This shows that current SAST practices in EMBOSS are not adequate.

Not Using SAST Tools. 20% (5/25) of the developers use no SAST tools. Most of these respondents (3/5) believe the security vulnerabilities in the corresponding projects have a low impact. However, these projects have an average OSSF criticality score of 0.43 ("high"). These developers may underestimate the severity of security issues in their projects, in line with previous studies [61, 106]. The developers of another project reported insufficient resources (*e.g.,* time). Unfortunately, this project is one of the most popular (>5K stars) open-source C++ libraries for embedded systems, with an OSSF score of >0.6 ("critical").

A final common reason for non-SAST use was concern about their effectiveness. Five respondents felt that using SAST tools on embedded software is questionable and might result in many false positives. This finding is consistent with previous surveys of non-embedded software developers [56].

To summarize our findings from this study of SAST prevalence in EMBOSS:

Finding 1: Sophisticated SAST tools are rarely used in EMBOSS repositories. According to our workflow analysis, only 4% of the EMBOSS repositories do so. With the same measure, 19% of non-embedded OSS do. Many EMBOSS repositories rely only on compiler warnings for SAST, which fail to find many common security defects.
Finding 2: The surveyed developers are generally aware of CI workflows and use them to run their SAST tools. When they do not use SAST, it is commonly because they believe the security impact or effectiveness of SAST is low.

5 SOTA SAST Performance

Given the lack of SAST tool usage in EMBOSS, our aim is to understand the effectiveness of SOAT SAST on EMBOSS.

5.1 Selection of the SOTA SAST Tools

Table 2. Comparison of the SAST tools we considered, on the Juliet benchmark and our EMBOSS dataset. The median # of warnings is reported for repositories where the tool ran successfully (CodeQL did not produce warnings on over half of repositories). *EMBOSS dataset precision is estimated via sampling. The cpp-lint performance measurement may be biased (see text).

GitHub Action	SAST	Juliet perf.	# Repos	Failure(s)	Med. # warn.	Precision
david-a-wheeler/flawfinder	flawfinder [101]	Error	176 (68%)	Invalid SARIF; Crashes	12	64/316 (20%)
cpp-linter/cpp-linter-action	cpplinter [57]	Timeout	230 (89%)	Timeout; Crashes	111	0/213 (0%)*
deep5050/cppcheck-action	cppcheck [80]	Timeout	256 (99%)	Timeout	19	116/200 (58%)
github/codeql-action	CodeQL [18]	F_1: 0.21	74 (29%)	Autobuild failure	0	154/160 (**96%**)

We want to apply the best-performing open-source SAST tool that (1) has a low false positive/negative rate; (2) can be readily used on OSS repositories; (3) is stable (not pre-release), free (not requiring licenses), and "plug-and-play" (supports a range of compilers and does not require knowledge of program semantics/modeling, etc.).

Competing Tools. We selected popular GitHub Actions that perform SAST on C/C++ repositories, as shown in Table 2. Selection criteria are detailed in Our Extended Report [91]. We made a workflow for each Action to apply them uniformly to the benchmarks for comparison.

Juliet Benchmark Performance. As one measure of effectiveness, we tested these tools on the Juliet Test Suite. The Juliet Test Suite is a labeled dataset commonly used to test SAST tools [77]. It does not focus on embedded software, so this is a measure of performance on general C/C++ code that may not reflect performance on embedded code. We used a time limit of 6 hours, the maximum time allowed for a job on many CI platforms, such as GitHub CI [47].

The middle column of Table 2 shows the results. Most tools either errored out or timed out. CodeQL completed in 40 minutes. CodeQL raised 11,101 warnings with a precision of 71% (7,904/11,101) and a recall of 12% (7,904/65,263).[1]

EMBOSS Sample Performance. To obtain another vantage, we also ran the tools on the 258 repositories in our EMBOSS corpus. We needed ground truth to evaluate the precision of each tool. CodeQL produced 471 warnings, while the others produced between 4K-200K warnings. It was infeasible to check them all. Therefore, we randomly sampled warnings to check. Specifically, for each tool, we randomly selected 30 repositories with < 20 warnings and manually checked each warning for those repositories.

The final columns of Table 2 show results. CodeQL *has the highest precision by far, at 96% – unsurprising given its effectiveness on Juliet Test Suite.* cppcheck and flawfinder had false positive rates > 40%. We recorded cpp-linter as having 100% false positives. This was likely a flaw in our sampling approach: all sampled warnings were related to compile-time issues that did not cause cpp-linter to error out, but we expect the sampling approach caused us to only examine warnings related to projects it struggled to compile.

Given that CodeQL is the most effective tool, we next perform a thorough evaluation of CodeQL's effectiveness on EMBOSS.

5.2 Effectiveness of CodeQL

As shown in Table 2, CodeQL failed to run on 71% of EMBOSS repositories. Specifically, the `Autobuild` phase of CodeQL failed to handle the diverse build setup of these repositories. We, therefore, manually created build scripts for all repositories based on their documentation and existing CI workflows.

Build Scripts Creation. We made the build scripts cover as much of the codebase as possible (*e.g.,* by compiling all example applications and all supported architecture and boards whenever possible). We successfully created build scripts for 156 (60%) repositories. For the other 102 repositories, the build instructions were either missing (17), too complex (*i.e.,* unavailable toolchains or dependencies) (49), or we could not get them to work (36). This manual process took ~45-60 minutes per repository.

Analysis and Configuration Details of CodeQL. CodeQL supports many suites (*i.e.,* collections of queries). There are three built-in suites for security scanning: `default`, `cpp-security-extended`, and `cpp-security-and-quality`. Each is a subset of the next, so we used the largest of these, `cpp-security-and-quality`, which contains 166 queries. Despite the queries' effectiveness on non-embedded codebases, Our preliminary analysis showed that (i) A few of these queries are not applicable to embedded

[1]We count a reported flaw as a true positive if the reported location matches that of a ground truth bug.

software, and (ii) The risk of corresponding defects is low because of the lack of process support and OS abstractions. We identified nine such queries and excluded them from our analysis. Table 3 shows the list of queries and the corresponding rationale for their exclusion.

Table 3. Reasons for excluding certain CodeQL queries. "Code readability" means the query detects code readability issues but not defects.

Query	**Reason for ignoring**
cpp/path-injection	Inapplicable
cpp/world-writable-file-creation	Inapplicable
cpp/poorly-documented-function	Code readability
cpp/potentially-dangerous-function[2]	Low-risk (Lack of OS abstractions and arbitrary process support)
cpp/use-of-goto	Code readability
cpp/integer-multiplication-cast-to-long	Low-risk (Most embedded device configurations are 32-bit)
cpp/comparison-with-wider-type	Low-risk (Most embedded device configurations are 32-bit)
cpp/leap-year/*	Low-risk
cpp/ambiguously-signed-bit-field	Low-risk

Furthermore, based on initial results, we modified three queries to improve their precision and ignore certain restrictions. First, we modified the `cpp/stack-address-escape` query to ignore cases of assigning a function parameter of a pointer type to a non-local variable. This usage is commonplace in practice and is unlikely to constitute a defect of significant concern as embedded systems usually have a fixed memory layout. Second, we modified `cpp/constant-comparison` to only report comparisons that are always false because we found that always-true comparisons are usually not defects in the EMBOSS context. For instance, developers can be overly cautious and perform the same check multiple times, where the second check will always be true. *e.g.,* `... if (p != NULL) { ... if (p != NULL) ... }`. Third, we modified `cpp/uninitialized-local` to eliminate false positives caused by casting a variable explicitly to `void`. Developers prevalently use such casts in EMBOSS to suppress compiler warnings on unused variables, *e.g.,* `(void) x;`. CodeQL accepted this third modification into their main repository [88]. We did not propose the first two modifications to the CodeQL team, as they may increase the false negative rate in general-purpose software analysis and are more appropriate as domain-specific refinements for embedded systems.

Finally, we created GitHub workflows for the 156 successfully-built repositories. These workflows invoke the necessary build scripts and run CodeQL with the required configuration. We ran these GitHub workflows on these 156 repositories. This produced many *issues*, which CodeQL divides into *errors* (high-severity concerns, *e.g.,* memory un-safety) and *warnings* (lower-severity issues, *e.g.,* code smells).

Table 4 shows the summary of our results across all repositories. We discuss the results by answering the following research questions:

RQ1 What defects does CodeQL find in EMBOSS?
RQ2 How do results vary by EMBOSS type?
RQ3 What is the false positive rate of CodeQL?
RQ4 How do developers respond to CodeQL results?
RQ5 Will developers integrate CodeQL in CI pipelines?

[2]The query "cpp/potentially-dangerous-function" checks for calls to gmtime, localtime, ctime and asctime. These functions are not thread-safe.

Table 4. Summary of CodeQL results: setup, raw data, manual analysis, responsible disclosure, and workflow integration.

Number of ...	Value	Number of ...	Value
Setup		**Responsible Disclosure**	
Repos in dataset	258	Defects disclosed	586
Repos built	156	Defects confirmed	376
Repos analyzed	151	Security defects disclosed	433
CodeQL **Results**		Security defects confirmed	302
Errors reported	772	Patch PRs submitted	163
Warnings reported	2,286	Patch PRs merged (*i.e.,* accepted)	104
Manual Analysis		CVEs issued	2
Defects discovered	709	CodeQL **SAST Workflow**	
Repos with defects	97 (64%)	Workflow PRs submitted	129
Security defects	535	Workflow PRs merged	37 (71% Active) and 29% Total)
Repos with security defects	85 (56%)		

5.2.1 RQ1: Defects Identified by CodeQL *in EMBOSS.* We manually analyzed all CodeQL issues for 151 repositories (out of a possible 156). The others have a substantial number of issues, and we did not have time to analyze them thoroughly. As reported in the *Defects Discovered* row of Table 4, we identified 709 defects across 97 repositories. We also distinguish the proportion of defects that can be deemed security-relevant to understand whether the studied CodeQL query suite has security benefits (*e.g.,* memory safety issues) vs. broader quality benefits (*e.g.,* code smells). To be conservative, we define a defect as **security-relevant** if and only if (1) the CodeQL query finding the defect contains the security tag, or (2) the defect is clearly related to memory safety (*e.g.,* null pointer dereference, out-of-bounds read/write). There were 535 (85 repositories) security-relevant defects, including in major projects maintained by organizations like Microsoft, Amazon, and the Apache Foundation. EMBOSS engineers have confirmed 376 (53%) of these defects, mainly by accepting our pull requests.

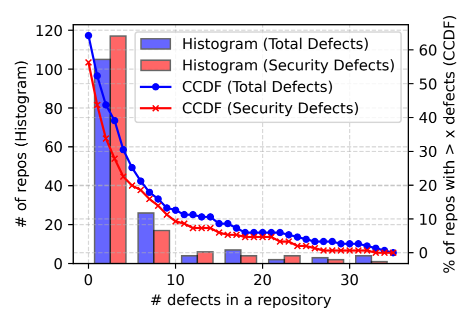


Fig. 3. Histograms (left axis) and CCDFs (right axis) of # of total and security-relevant defects in a repository. The CCDF values at 0 defects indicate that defects are discovered in 64% of repositories, and security-relevant defects are discovered in 56% of repositories.

Defect Rates Per Repository. Figure 3 shows Complementary Cumulative Distribution Functions (CCDFs) of the number of total defects and security defects in each repository.[3] The left-most point on both the lines indicates that there are 64% (97) repositories with at least one defect, and 56% (85) repositories with at least one security defect. The security defects line has almost the same trend as total defects, which is consistent with our finding that a large proportion (535 out of 709, or 75%) of the identified defects are security-relevant. Although ~90% of the repositories have less than ten total defects, nine repositories have significantly more. Table 5 lists the top 5 repositories with the most total defects and their criticality scores.

[3]A point (x, y) on a CCDF [103] line indicates that y% of repositories contain greater than x corresponding type of defects.

Common Types of Security Defects. We found several classes of security defects across all repositories. Figure 4 shows the top 10 types of security-relevant defects [46] found along with the corresponding number of defects. We discuss three main types of security defects below. Code examples for each type are provided in Our Extended Report [91].

Table 5. Top-5 EMBOSS repositories by number of total defects found.

Repo	Criti. Score	# Total	# Security
apache/nuttx	0.69	35	24
contiki-ng/contiki-ng	0.67	34	24
raysan5/raylib	0.70	33	33
ARMmbed/mbed-os	0.72	32	22
openlgtv/epk2extract	0.45	29	27

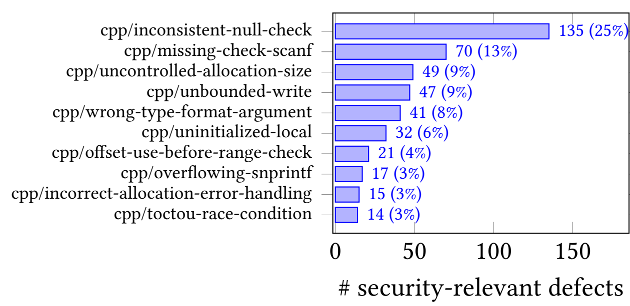


Fig. 4. Top-10 CodeQL queries by security-relevant defects found.

• *cpp/inconsistent-null-check*. This rule identifies cases where the return value of a function is not checked for `NULL`, despite most other calls to the same function performing such a check. Developers should consistently validate return values that may be `NULL` to prevent potential null pointer dereferences. This rule flagged 135 such instances. This rule detected 135 such instances.

• *cpp/uncontrolled-allocation-size*. This rule detects cases where the size argument of a memory allocation function (*e.g.,* `malloc`) is computed through integer arithmetic involving potentially untrusted input (*e.g.,* user input). If the input takes on large values, an integer overflow [38] may occur, resulting in an allocation size significantly smaller than intended. Subsequent buffer accesses may lead to out-of-bounds reads or writes. This rule identified 49 such instances.

• *cpp/unbounded-write*. This rule detects out-of-bound write vulnerabilities. Specifically, this includes analysis of potentially dangerous function calls (*e.g.,* `strcpy`, `sscanf`) to check whether these are used properly with valid arguments. This rule detected 47 vulnerabilities of potential buffer overflow.

Severity of Security Defects. The severity of a security bug depends on its exploitability and the criticality of the underlying software [33, 93]. Given the large number of defects, manually assessing exploitability is intractable. Instead, we use the OSSF criticality score (§2.1.2) of the target repository to assess the severity of a bug. Figure 5 shows the CCDF of the severity of security defects. Specifically, a point (x, y) on the line indicates y% of the defects have severity greater than x. Approximately 50% of bugs have a severity score of more than 0.5, which represents high-severity repositories (§2.1.2). Specifically, ~40% of bugs have a score of more than 0.6, representing vulnerabilities in critical repositories. For instance, we found an arbitrary write vulnerability in `Mbed-TLS/mbedtls` (criticality score = 0.73, Listing 1) and a use-after-free in `apache/nuttx` (criticality score = 0.69, Listing 2); both of these are critical projects.

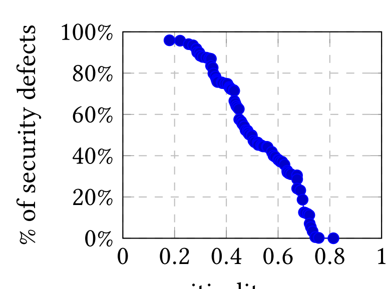


Fig. 5. CCDF of the severity of security defects.

```c
// mbedtls/programs/ssl/ssl_mail_client.c
int main(int argc, char *argv[]) {
  unsigned char buf[1024];
  char *q = strchr(argv[1], '=') + 1;
  opt.mail_from = q;
  len = sprintf((char *) buf, "MAIL
↪  FROM:<%s>\r\n", opt.mail_from);
  ...
}
```

Listing 1. Unbounded `sprintf` formats attacker-controlled `opt.mail_from` into the fixed-size stack buffer `buf`, enabling a classic stack buffer overflow. (Code is simplified for clarity.)

```c
// apache/nuttx/drivers/sensors/apds9960.c
ret = register_driver(devpath,
↪  &g_apds9960_fops, 0666, priv);
if (ret < 0) {
  snerr("ERROR: Failed to register driver:
  ↪  %d\n", ret);
  kmm_free(priv);
}
priv->config->irq_attach(priv->config,
↪  apds9960_int_handler, priv);
```

Listing 2. The memory pointed by `priv` is freed inside the `if` condition. It is accessed later on, resulting in use-after-free.

Common Types of Non-Security Defects. These defects may not lead to security vulnerabilities but can cause functionality issues, undefined behavior, and compilation issues. For instance, the rule `cpp/missing-return` detects non-void functions with no explicit return statement. This may result in undefined behavior during runtime [59]. Similarly, the rule `cpp/virtual-call-in-constructor` detects calls to virtual functions in a constructor. This also could lead to undefined behavior as the object's virtual table may not be completely initialized [48]. Figure 6 shows the top ten non-security defects along with the corresponding number of defects.

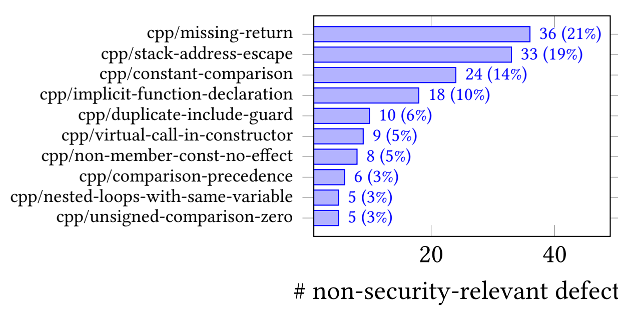


Fig. 6. Top-10 CodeQL queries by non-security defects found.

5.2.2 RQ2: Trends by EMBOSS Type. Figure 7 shows the number of defects found across various repositories according to their categories. *At a high level, across all categories, the number of security defects is more than that of the number of non-security defects.* Furthermore, the number of defects is proportional to the number of repositories of the particular category (Table 1). For instance, Network (NET), Operating Systems (OS), and Applications (APP) are the top three categories containing the highest number of repositories (128 (50%)), and they also contain the highest number of defects (423 (60%)). The Memory Management Libraries (MML) with the least number (4) of repositories also have the least defects (6). Interestingly, we noticed that defect density, *i.e.,* number of defects per KSLOC, is non-uniform. Our Extended Report [91] provides defect distribution per-repository and defect density across various categories of EMBOSS. In summary, APP and NET have the highest defect densities. On the other hand, OS and HAL have the lowest densities. *Our results empirically show defect density is not uniform across different categories of EMBOSS.*

5.2.3 RQ3: False Positive Rates. False positive rate analysis requires a significant amount of work, yet false positives are also a major concern in the adoption of SAST tools. Given the large number of repositories, we sampled 123 successfully built repositories (the 50 most starred, the 50 least starred, and 23 randomly picked repositories) and manually categorized all issues in them into true and false positives. A false positive means that the result does not match what the rule intends to detect, *e.g.,* an error for an uninitialized variable when it is actually initialized. Two analysts worked

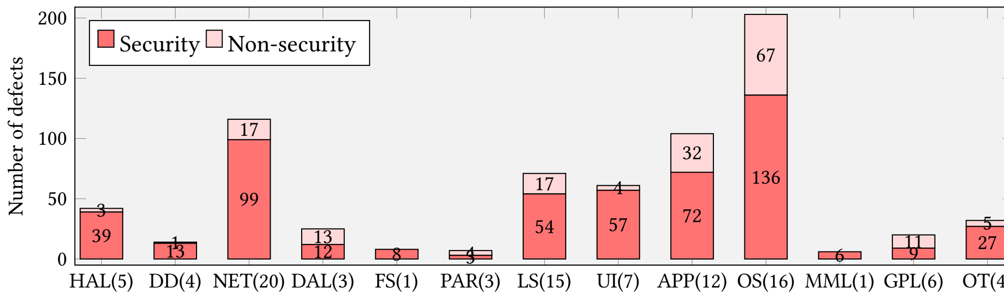


Fig. 7. Number of defects found in repositories of each category (defined in Table 1). On x-axis, numbers show # repositories with ≥ 1 defect.

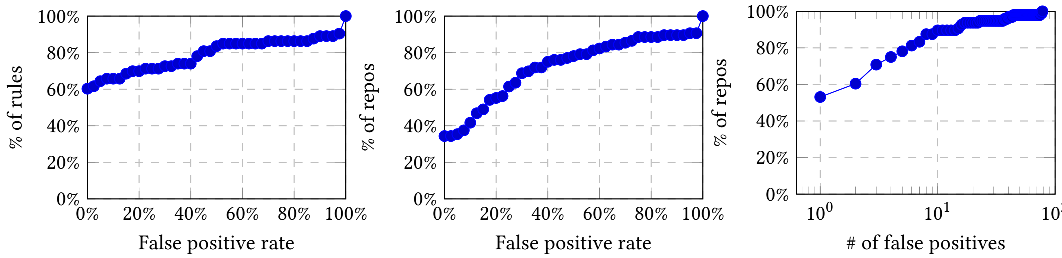
for one month to analyze the results. The two analysts worked largely independently but discussed uncertainties with each other and with the rest of the research team. All analysts and researchers had substantial training (coursework and experience) in C/C++ programming and cybersecurity, and we believe they were able to make correct judgments about whether or not a CodeQL issue represented a true positive.

The overall percentages of true and false positives are 66% (1039/1577) and 34% (538/1577), respectively. Figure 8a shows the Cumulative Distribution Function (CDF) of the false positive rates of different rules. Specifically, a point (x, y) on a line indicates y% of the rules have false positive rates of less than or equal to x%. Approximately 60% of the rules had no false positives, and 10% had no true positives. This indicates that false positives are polarized, and a few rules contribute to the majority of false positives. Specifically, 20% of rules contribute to more than 60% of false positives. We discuss below the top four CodeQL queries contributing to false positives. Comprehensive information on all rules contributing to false positives is provided in Our Extended Report [91].

- *cpp/uninitialized-local.* Dataflow analysis of CodeQL is not path-sensitive. Some variables may not be initialized in all paths. However, when a variable is used, certain path conditions hold, under which it can be proved that the variable must have been initialized.
- *cpp/missing-check-scanf.* Developers can use `switch case` statements (instead of `if` statements) to check the return value of `scanf` calls. These are valid checks but not detected by CodeQL.
- *cpp/suspicious-pointer-scaling* and *cpp/suspicious-pointer-scaling-void.* These rules detect risky pointer arithmetic operations. However, pointer casts, and type-punning are pretty common and unavoidable in low-level embedded system code.
- *cpp/unbounded-write.* `strcpy` is safe if the destination must be large enough. For example, developers can first use `strlen` to calculate the length of the source string, allocate enough memory for the destination string, and then call `strcpy`.

Although the cumulative false positive rate is high (34%), it does not affect most repositories. Figure 8b shows the CDF of % of repositories and false positive rate; we can see that ∼40% of repositories have no false positives and more than 60% of the repositories have less than 20% false positive rate. Furthermore, *the actual number of false positives is very low, as shown in Figure 8c. Specifically, ∼55% of the repositories have less than one false positive, and 90% of repositories have less than ten false positives.* These results show that the majority of EMBOSS repositories are not affected by false positives.

5.2.4 RQ4: Developer Response on Defects Identified by CodeQL. We responsibly disclosed all identified defects in repositories that are actively maintained (had commits in the past three months). We opened issues and raised pull requests with appropriate patches where possible. The bottom



(a) CDF of the false positive rates of rules

(b) CDF of the false positive rate vs. % of repos

(c) CDF of # of false positives vs. % of repos

Fig. 8. CDF of the false positive rates of rules, along with CDFs of the rate and number of false positives vs. percentage of repositories (repos).

of Table 4 summarizes the developer response. In total, 53% (376/709) of defects have been confirmed by developers (via merging our pull requests or expressing confirmation in replies to issues).

Most of the patches were readily accepted by the developers. In a few cases, developers were even interested in knowing the techniques we used to find the defects. For instance, developers of an AWS-owned repository said *"I'm curious how you stumbled across this – Was there some sort of test you ran or was this something that came up during your development? I'm hoping we can duplicate your method of discovery to add some sort of check/test to the repo."*

There were two pull requests where the developers did not choose to fix potential security issues. They stated that although code robustness is important, they deemed reduced code size and RAM usage to be a higher priority in their embedded software. These observations support the conventional wisdom that software engineers (and especially engineers in embedded systems) trade-off between security and performance [43, 50].

Although many security-relevant defects were resolved, only two Common Vulnerabilities and Exposures (CVEs) were assigned. When we disclosed the security-relevant defects, we did not explicitly ask the engineering teams to issue CVEs. Of the 94 repositories against which we opened at least one security-relevant defect, only two issued CVEs for these defects: `mbedtls` issued CVE-2023-29472, and `contiki-ng` issued CVE-2023-30546. We eventually followed up on our 77 reports of defects to the 10 most popular repositories (by GitHub stars) to inquire whether CVEs were being prepared. Two of the engineering teams replied suggesting that we email their security teams – we did so, but received no response. The other eight teams did not respond. Our research supports the observation of prior work [65], that security defects are often fixed "silently", without tracking via a CVE.[4]

5.2.5 RQ5: Developer Response on Integrating CodeQL *Workflows.* We opened pull requests to integrate our CodeQL scanning workflows into the CI pipeline of the projects. This would have the effect of using CodeQL's SAST to check all subsequent pull requests. We measured the number of merged pull requests and the kinds of replies made by the developers. An example pull request is given in Our Extended Report [91].

We raised 129 pull requests to integrate our CodeQL workflows into the corresponding projects. We did not submit some pull requests as the repositories do not accept external contributions, *e.g.,* Microsoft Azure. In addition, some of our workflows became out of date due to concurrent changes in the project's build process. Our Extended Report [91] shows our pull request. We received

[4]With the recent threat to funding for the CVE system [86, 99], this finding provides further evidence that CVEs may not be the best metric for assessing the impact of a work.

responses for 52 of our pull requests, of which 37 were merged (71% acceptance rate for responses, 29% acceptance rate overall).

Accepted Requests. Most of the developers readily accepted our workflow. In a few cases (3), we had to make syntactic adjustments to our workflow according to the repository coding practices. Few developers (2) had concerns of the effectiveness of CodeQL. When asked, we pointed to the defects we identified as evidence. Interestingly, one developer surveyed their friends on X (formerly Twitter) for opinions about CodeQL, before accepting our pull request.

Closed Requests. Several developers (7) closed our pull requests, assuming that these were generated by bots. We contacted them again to clarify that we were not bots but received no response. A few developers (3) mentioned that they do not have enough resources to handle the alerts raised by CodeQL. A few developers (2) mentioned concerns about licensing.

In summary, this part of our investigation yielded the following findings:

Finding 3 (RQ1): CodeQL finds hundreds of real defects in the studied EMBOSS repositories, including in repositories maintained by reputable organizations like Amazon and Microsoft.
Finding 4 (RQ2): Defect density (defects per SLOC) is not uniform across different categories of EMBOSS. Some categories of projects (*e.g.,* APP and NET) are more likely to contain defects than others (*e.g.,* OS and HAL).
Finding 5 (RQ3): CodeQL has a false positive rate of 34% in the 123 sampled repositories. However, false positives are polarized, *i.e.,* a few rules contribute to the majority of false positives.
Finding 6 (RQ3): Although the overall false positive rate is high, it has minimal impact on EMBOSS repositories: ~40% of repositories have no false positives, ~55% of the repositories have ≤ 1 false positive, and 90% of repositories have ≤ 10 false positives.
Finding 7 (RQ4): Developers readily accept fixes for defects identified by CodeQL – demonstrating that they care about these defects.
Finding 8 (RQ5): Many EMBOSS developers are willing to integrate the CodeQL SAST into their projects' CI as a GitHub workflow, provided that someone else (our research team) prepares, validates and explains the workflow for them.
Finding 9: A default Autobuild fails on many EMBOSS projects. However, producing a customized build suitable for CodeQL takes minimal engineering effort for developers.

6 Lessons Learned

We summarize our experiences in **four lessons** on using SAST in EMBOSS.

(Lesson 1) EMBOSS can benefit from SAST: Despite developers' misgivings about the effectiveness of SAST on EMBOSS, we found many security defects (535) across various embedded software by using an existing SAST tool. Developers acknowledged and fixed most of the security defects (70%) found by SAST tools, which shows that SAST tools can find important defects. Since many of these repositories (96%) did not use SAST tools, it is perhaps unsurprising that they were rife with defects that SAST can detect. Nevertheless, *evidence* of this is important to push the EMBOSS engineering community toward more responsible engineering practice.

(Lesson 2) Developers are willing to adopting SAST in EMBOSS Repositories: Several developers accepted our pull requests (71% (Active) and 29% (Total)) to integrate a SAST tool (*i.e.,* CodeQL) into their CI pipeline. Our pull request was well-formatted and included all the necessary details along with evidence of CodeQL's effectiveness. Specifically, we included the examples of the defects found by CodeQL in the corresponding repository. Furthermore, there was not much persuasion needed to accept our pull requests. We draw two sub-lessons here.

First, engineers can easily integrate SAST tool into EMBOSS repositories – the pull request is not too complex and can be done without much project-specific expertise. Second, engineers will accept contributions from researchers, provided the contributions come with a demonstration of effectiveness (*i.e.,* an acceptable cost-benefit tradeoff).

(Lesson 3) SAST tool developers should consider the properties of EMBOSS: Our experience shows that certain SAST queries, which are effective on traditional (non-embedded) codebases, might be ineffective or inapplicable for embedded codebases. We therefore recommend that SAST tool developers take the characteristics of embedded codebases into consideration while evaluating their tool design decisions. Part of our contribution is a set of modifications and configurations of CodeQL queries that demonstrate the kinds of changes that are needed.

(Lesson 4) We need more best-effort defect detection techniques for EMBOSS: We were able to find a large number of defects (709), including security vulnerabilities (535), in EMBOSS repositories by just using an off-the-shelf SAST tool. Our results complement a recent work [12] that used simple systematic testing to find several severe security issues in popular EMBOSS network stacks. These works provide strong evidence that the EMBOSS engineering community should investigate the potential of integrating simple or best-effort defect detection techniques.

7 Future Work

Developers accepted our pull requests to integrate a SAST tool (*i.e.,* CodeQL) into their CI pipeline. However, this required manual effort (although minimal) to identify the build setup, create a CI workflow, and raise the pull request. As part of our future work, we plan to automate this process by using Large Language Models (LLMs) assisted techniques [107]. To further encourage the adoption of SAST tools, we plan to create rewards badges (such as the OpenSSF Best Practices Badge [95]), or public recognition for projects that demonstrate the successful use of SAST tools in finding and fixing vulnerabilities. We also plan to create tutorials, workshops, and documentation that showcase the effectiveness of SAST tools in identifying real-world vulnerabilities that can help EMBOSS developers better understand their value.

8 Limitations and Threats to Validity

Like any empirical study, our study has a range of limitations. We distinguish three types of threats to validity [104]. Guided by Verdecchia *et al.*, we focus on substantive threats that might influence our findings [98].

Construct Threats are potential limitations of how we operationalized concepts. We scope the construct of security vulnerabilities to those detectable by the SAST tools from the GitHub Marketplace, particularly those captured by CodeQL queries with the security tag or those associated with memory safety. Other classes of security vulnerabilities, and other kinds of software defects, are beyond the scope of our work.

Internal threats are those that affect cause-effect relationships. This work was primarily a measurement study, which does not involve causal inferences. However, our motivation stemmed in part from the observation that many EMBOSS projects do not use SAST, and that the surveyed developers often cited the perceived complexity and noisiness of applying SAST. Our measurements are thus useful in shaping software engineering practice only insofar as these statements are truthful.

External threats may impact generalizability. Here is where most of the threats are.

- *Focus on free SAST tools:* We applied SAST tools available in the GitHub Marketplace to the open-source embedded software available on GitHub. Our results may not generalize to other SAST tools, particularly commercial ones such as Coverity and Sonar.

- *Focus on EMBOSS:* Our results may not generalize to other embedded software, particularly commercial embedded software, to which costly techniques such as formal methods may have been applied [11, 13]. To shed some light on this threat, in our analysis, we showed that a SAST tool (*i.e.,* CodeQL) was still able to find defects in commercially-developed open-source software, such as Amazon's `aws/aws-iot-device-sdk-embedded-C` (which uses the commercial Coverity SAST tool).
- *Scoping to GitHub:* Our study may suffer from data collection bias as we focus on projects and SAST tools available on GitHub. There could be other EMBOSS projects (*e.g.,* in BitBucket) and tools on which our observations may not hold. We tried to avoid this by collecting diverse projects with varying sizes.
- *Limited developer study:* Given the low number of responses, the observations from our developer study (§4.3.2) may not generalize to other EMBOSS repositories. As a modest mitigation, we note that the response rate was consistent with other surveys of GitHub developers [2, 55].

9 Related Work

Earlier we discussed directly related work. Here we compare broadly.

Embedded Operating Systems and Frameworks: Al-Boghdady *et al.* [3] conducted a thorough analysis of four IoT Operating Systems, namely RIOT, Contiki, FreeRTOS, and Amazon FreeRTOS. Their results indicated an increasing trend in the number of security errors over time. Others agreed: Alnaeli *et al.* [6, 7] reported a rise in unsafe statements in Contiki and TinyOS, and McBride *et al.* [72] found increasing error rates in Contiki. Malik *et al.* [66] shed some light on root causes, noting that the complex behaviors of embedded devices are challenging to validate internally. Our work encompasses these OSes and includes a wider range of embedded software, leading to a broader view of the state of EMBOSS.

Other Analyses of Embedded Systems: Embedded software has been studied for decades. We highlight a few recent analyses. Peng *et al.* [81] proposed a CI environment to improve the efficiency and quality of software development in the nuclear power industry. The XANDAR project [39] combines a model-based toolchain and hypervisor-based runtime architecture to create embedded software systems with safety, security, and real-time properties. Bagheri *et al.* [20] proposed a method for automatically generating assurance cases for software certification. Jia *et al.* [54] used control and data flow analysis to find malicious behavior in IoT applications. Celik *et al.*'s SOTERIA system [26] combines static analysis and model checking to find security and safety violations in IoT software. Complementing these studies, we focused on static analysis for embedded software to understand current practices, challenges, and opportunities.

Developers' Perspectives on SAST Tools: For SAST, many works have examined the factors hindering or spurring adoption. Johnson *et al.* [56] found that false positives and (non-)usability of warnings are barriers. More recently, Ami *et al.* [9] interviewed 20 practitioners and found that they considered these tools to be highly beneficial complements to manual analysis. Among the challenges faced by developers, the significant pain points were false negatives, the absence of meaningful alert messages, and the effort required for configuration and integration. Wadhams *et al.* [100] identified false positives, poor output, time-consuming setup, and manual effort for fixes as the primary barriers to SAST adoption. They emphasized that both developers and SAST tool creators have distinct yet equally crucial roles in promoting the widespread use of SAST. Our study revealed slightly different findings. In addition to false positives, developers were unaware of the effectiveness of SAST tools on embedded software.

In terms of tool performance, Lenarduzzi *et al.* have questioned the tools' capabilities [58], comparing six SAST tools for Java and finding little agreement among them as well as low precision. Our experience contrasts with their findings. Our experiments with CodeQL demonstrate that SAST tools are highly capable of identifying vulnerabilities within the EMBOSS context. We were able to easily (with minimal engineering effort) configure and integrate CodeQL in EMBOSS repositories. The alert messages were displayed in SARIF format and were easy to understand and evaluate.

10 Conclusions

We evaluated the usage and effectiveness of SAST in EMBOSS. Across 258 open-source embedded software projects, the CodeQL SAST tool found 709 defects (with a false positive rate of 34%), 376 of which have been confirmed. These included 302 defects that were security vulnerabilities such as crashes and memory corruption. False positives were mainly caused by a few outlier CodeQL rules and projects. For the majority of repositories studied, the false positive rates were low. We also raised pull requests to incorporate our CodeQL workflows as part of EMBOSS CI pipeline, out of which 37 (71% (Active) and 29% (Total)) are already accepted. We conclude that the current generation of static analysis tools, exemplified by CodeQL, has overcome concerns about false positives and can be easily incorporated into embedded software projects. If engineers adopted these tools, many security vulnerabilities would be prevented. *Future research should push the bounds of vulnerability discovery, but we call for efforts to promote adoption of existing tools.*

11 Research Ethics

In the conduct of this study, we upheld two ethical duties: the responsible conduct of research on human subjects, and the appropriate handling of cybersecurity vulnerabilities.

Ethics for human-subjects research: Studies of human subjects must offer a favorable risk-reward tradeoff. Our study included a human-subjects study: we surveyed EMBOSS software engineers. The possible risk to our subjects was professional scrutiny based on following (or not following) best practices in software engineering such as using SAST. The benefit is an increased awareness of the available SAST tools and their performance, which may benefit them directly, as well as those who depend on their software, and the broader EMBOSS community.

This study was conducted with the approval of our institution's Institutional Review Board (IRB).

Ethics for cybersecurity vulnerabilities: The ethical duty for handling cybersecurity vulnerabilities requires responsible disclosure to protect users and systems by informing relevant parties about identified security risks in a timely manner. Responsible disclosure typically follows one of two models: Coordinated Vulnerability Disclosure (CVD) or Full Disclosure. CVD involves informing the responsible parties first, allowing them time to address the issue before publicizing the vulnerability. Full Disclosure, in contrast, is the practice of publishing vulnerability analyses as soon as possible, without a private coordination period with the affected project or organization.

Since we identified vulnerabilities without associated exploits, and these vulnerabilities could be found by anyone applying a SOTA tool, we determined that secrecy was not necessary. We therefore adopted the Full Disclosure approach by opening public issues or pull requests with patches for the identified vulnerabilities, as detailed in §5.2.4. By supplying a patch alongside the vulnerability report, we actively mitigated the risk to users by making it easier for maintainers to address the issue promptly. EMBOSS engineers frequently fixed the vulnerabilities we identified, and none raised concerns that our public reporting was unethical.

Data Availability

All project data are available at https://github.com/purs3lab/ISSTA-2025-EMBOSS-Artifact and [90].

Acknowledgments

This research was supported by the National Science Foundation (NSF) under Grants CNS-2340548 and CNS-2247686 and Rolls-Royce Grant on "Dynamic Analysis of Embedded Systems." The U.S. government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notation thereon. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF or Rolls-Royce.

References

[1] [n. d.]. OSRTOS. https://www.osrtos.com/.

[2] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L. Mazurek, and Sascha Fahl. 2017. Security Developer Studies with GitHub Users: Exploring a Convenience Sample. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*. USENIX Association, Santa Clara, CA, 81–95.

[3] Abdullah Al-Boghdady, Khaled Wassif, and Mohammad El-Ramly. 2021. The Presence, Trends, and Causes of Security Vulnerabilities in Operating Systems of IoT's Low-End Devices. *Sensors* 21, 7 (2021). doi:10.3390/s21072329

[4] Mohammed Ali Al-Garadi, Amr Mohamed, Abdulla Khalid Al-Ali, Xiaojiang Du, Ihsan Ali, and Mohsen Guizani. 2020. A Survey of Machine and Deep Learning Methods for Internet of Things (IoT) Security. *IEEE Communications Surveys & Tutorials* 22, 3 (2020), 1646–1685. doi:10.1109/COMST.2020.2988293

[5] Fadi Al-Turjman and Joel Poncha Lemayian. 2020. Intelligence, security, and vehicular sensor networks in internet of things (IoT)-enabled smart-cities: An overview. *Computers & Electrical Engineering* 87 (2020), 106776.

[6] Saleh M. Alnaeli, Melissa Sarnowski, Md Sayedul Aman, Ahmed Abdelgawad, and Kumar Yelamarthi. 2016. Vulnerable C/C++ code usage in IoT software systems. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)* (Reston, VA, USA, 2016-12). IEEE, 348–352. doi:10.1109/WF-IoT.2016.7845497

[7] Saleh Mohamed Alnaeli, Melissa Sarnowski, Md Sayedul Aman, Ahmed Abdelgawad, and Kumar Yelamarthi. 2017. Source Code Vulnerabilities in IoT Software Systems. 2, 3 (2017), 1502–1507. doi:10.25046/aj0203188

[8] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1362–1380. doi:10.1109/SP.2019.00013

[9] Amit Seal Ami, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. 2024. "False negative - that one is going to kill you": Understanding Industry Perspectives of Static Analysis based Security Testing. In *2024 IEEE Symposium on Security and Privacy (SP)*. 3979–3997. doi:10.1109/SP54263.2024.00019

[10] Mahdi Amiri-Kordestani and Hadj Bourdoucen. 2017. A survey on embedded open source system software for the internet of things. In *Free and Open Source Software Conference*, Vol. 2017.

[11] Paschal C. Amusuo, Owen Cochell, Taylor Le Lievre, Parth V. Patil, Aravind Machiry, and James C. Davis. 2025. Do Unit Proofs Work? An Empirical Study of Compositional Bounded Model Checking for Memory Safety Verification. (2025). arXiv:2503.13762 [cs.SE] https://arxiv.org/abs/2503.13762

[12] Paschal C. Amusuo, Ricardo Andrés Calvo Méndez, Zhongwei Xu, Aravind Machiry, and James C. Davis. 2023. Systematically Detecting Packet Validation Vulnerabilities in Embedded Network Stacks. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 926–938. doi:10.1109/ASE56229.2023.00095

[13] Paschal C Amusuo, Parth V Patil, Owen Cochell, Taylor Le Lievre, and James C Davis. 2025. A Unit Proofing Framework for Code-level Verification: A Research Agenda. In *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*.

[14] Sundaram Ananthanarayanan, Masoud Saeida Ardekani, Denis Haenikel, Balaji Varadarajan, Simon Soriano, Dhaval Patel, and Ali-Reza Adl-Tabatabai. 2019. Keeping Master Green at Scale. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. 15 pages. doi:10.1145/3302424.3303970

[15] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*. 1093–1110.

[16] Andrei Arusoaie, Stefan Ciobâca, Vlad Craciun, Dragos Gavrilut, and Dorel Lucanu. 2017. A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 161–168. doi:10.1109/SYNASC.2017.00035

[17] Abhishek Arya, Caleb Brown, Rob Pike, and The Open Source Security Foundation. 2023. Open Source Project Criticality Score. https://github.com/ossf/criticality_score. original-date: 2020-11-17T16:14:23Z.
[18] Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*.
[19] AWS open source. 2024. *FreeRTOS – Real-time operating system for microcontrollers.* https://www.freertos.org/
[20] Hamid Bagheri, Eunsuk Kang, and Niloofar Mansoor. 2020. Synthesis of Assurance Cases for Software Certification. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*.
[21] Pranshu Bajpai and Adam Lewis. 2022. Secure Development Workflows in CI/CD Pipelines. In *2022 IEEE Secure Development Conference (SecDev)*. 65–66.
[22] Deval Bhamare, Maede Zolanvari, Aiman Erbad, Raj Jain, Khaled Khan, and Nader Meskin. 2020. Cybersecurity for industrial control systems: A survey. *Computers & Security* 89 (2020), 101677.
[23] Davide Bonaventura., Sergio Esposito., and Giampaolo Bella. 2023. Smart Bulbs Can Be Hacked to Hack into Your Household. In *Proceedings of the 20th International Conference on Security and Cryptography (SECRYPT 2023)*. doi:10.5220/0012092900003555
[24] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (Dec. 2018), 112–129. doi:10.1016/j.jss.2018.09.016
[25] Guillaume Brat, Jorge A Navas, Nija Shi, and Arnaud Venet. 2014. IKOS: A framework for static analysis based on abstract interpretation. In *Proceedings of the International Conference on Software Engineering and Formal Methods (SEFM 2014)*. Springer.
[26] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 147–158.
[27] CERN. 2023. Computer Security: Avoiding salmonella in your code. https://rb.gy/yky2e
[28] Chaminda Chandrasekara and Pushpa Herath. 2021. Hands-on GitHub Actions: Implement CI/CD with GitHub Action Workflows for Your Applications. (2021).
[29] George Chatzieleftheriou and Panagiotis Katsaros. 2011. Test-driving static analysis tools in search of C code vulnerabilities. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*. 96–103.
[30] Checkmarx Ltd. 2023. *Checkmarx*. https://checkmarx.com/
[31] Travis CI. 2023. Travis CI - Test and Deploy Your Code with Confidence. https://travis-ci.org/.
[32] Circle Internet Services, Inc. 2024. Continuous Integration and Delivery - CircleCI. https://circleci.com/.
[33] Roland Croft, M. Ali Babar, and Li Li. 2022. An Investigation into Inconsistency of Software Vulnerability Severity across Data Sources. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. doi:10.1109/SANER53432.2022.00050
[34] Fredrik Dahlgren. 2022. Trail of Bits finds bugs using CodeQL. https://blog.trailofbits.com/2022/01/11/finding-unhandled-errors-using-codeql/.
[35] Tobias Dam, Lukas Daniel Klausner, and Sebastian Neumaier. 2023. Towards a Critical Open-Source Software Database. In *Companion Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23 Companion)*. doi:10.1145/3543873.3587336
[36] Albert Danial. 2021. *cloc: v1.92*. doi:10.5281/zenodo.5760077
[37] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. 2022. On the use of GitHub actions in software development repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 235–245. doi:10.1109/ICSME55016.2022.00029
[38] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2015. Understanding integer overflow in C/C++. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 1–29.
[39] Tobias Dörr and et al. 2024. XANDAR: An X-by-Construction Framework for Safety, Security, and Real-Time Behavior of Embedded Software Systems. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. doi:10.23919/DATE58400.2024.10546852
[40] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. 2021. Sok: Enabling security analyses of embedded systems via rehosting. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21)*. doi:10.1145/3433210.3453093
[41] Daniel Feitosa, Apostolos Ampatzoglou, Paris Avgeriou, and Elisa Yumi Nakagawa. 2015. Investigating Quality Trade-Offs in Open Source Critical Embedded Systems. In *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures* (Montréal, QC, Canada) *(QoSA '15)*. 113–122. doi:10.1145/2737182.2737190
[42] Nicole Forsgren, Jez Humble, and Gene Kim. 2018. *Accelerate: The science of lean software and devops: Building and scaling high performing technology organizations.* IT Revolution.
[43] Radek Fujdiak, Petr Mlynek, Petr Blazek, Maros Barabas, and Pavel Mrnustik. 2018. Seeking the Relation Between Performance and Security in Modern Systems: Metrics and Measures. In *2018 41st International Conference on*

Telecommunications and Signal Processing (TSP). 1–5. doi:10.1109/TSP.2018.8441496
[44] Christoph Gentsch. 2020. Evaluation of open source static analysis security testing (SAST) tools for C. (2020).
[45] GitHub, Inc. 2021. CodeQL Wall of Fame. https://securitylab.github.com/codeql-wall-of-fame/.
[46] GitHub, Inc. 2023. CodeQL Query Help for C and C++. https://codeql.github.com/codeql-query-help/cpp/.
[47] GitHub, Inc. 2023. Usage limits, billing, and administration. https://rb.gy/kbnze2.
[48] GitHub, Inc. 2023. Virtual Call from Constructor or Destructor – CodeQL documentation. https://rb.gy/kkixfh.
[49] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. 2022. On the rise and fall of CI services in GitHub. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 662–672. doi:10.1109/SANER53432.2022.00084
[50] Nikhil Krishna Gopalakrishna, Dharun Anandayuvaraj, Annan Detti, Forrest Lee Bland, Sazzadur Rahaman, and James C. Davis. 2022. "If Security Is Required": Engineering and Security Practices for Machine Learning-based IoT Devices. In *4th International Workshop on Software Engineering Research & Practices for the Internet of Things (SERP4IoT)*. 8.
[51] Jez Humble and David Farley. 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation.* Pearson Education.
[52] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. 2019. How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)* (2019-10). 323–333. doi:10.1109/ISSRE.2019.00040
[53] Nasif Imtiaz and Laurie Williams. 2019. A synopsis of static analysis alerts on open source software. In *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security (HotSoS '19)*. Article 12, 3 pages. doi:10.1145/3314058.3317295
[54] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z. Morley Mao, and Atul Prakash. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *Network and Distributed Systems Security (NDSS) Symposium 2017*.
[55] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet Singh Kochhar, and Li Zhang. 2017. Why and how developers fork what from whom in GitHub. *Empirical Software Engineering* 22 (2017), 547–578.
[56] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *2013 35th International Conference on Software Engineering (ICSE)* (San Francisco, CA, USA). IEEE, 672–681. doi:10.1109/ICSE.2013.6606613
[57] Brenno Lemos. 2023. C/C++ Linter Action – Clang-Format & Clang-Tidy.
[58] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. 2023. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software* 198 (2023), 111575. doi:10.1016/j.jss.2022.111575
[59] Linearity. 2010. Omitting Return Statement in C++.
[60] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. 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 (ISSTA 2022)*. ACM, Virtual South Korea, 544–555. doi:10.1145/3533767.3534380
[61] Tamara Lopez, Helen Sharp, Thein Tun, Arosha Bandara, Mark Levine, and Bashar Nuseibeh. 2019. "Hopefully We Are Mostly Secure": Views on Secure Code in Professional Practice. In *2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 61–68. doi:10.1109/CHASE.2019.00023
[62] Bailin Lu, Wei Dong, Liangze Yin, and Li Zhang. 2018. Evaluating and integrating diverse bug finders for effective program analysis. In *Software Analysis, Testing, and Evolution (SATE 2018)*. Springer, 51–67.
[63] Björn Lundell, Brian Lings, and Anna Syberfeldt. 2011. Practitioner Perceptions of Open Source Software in the Embedded Systems Area. *Journal of Systems and Software* 84, 9 (2011). doi:10.1016/j.jss.2011.03.020
[64] Chujiao Ma, Matthew Bosack, Wendy Rothschell, Noopur Davis, and Vaibhav Garg. 2022. Wanted Hacked or Patched. (2022). https://www.usenix.org/sites/default/files/opensourcebugbounty_login_final.pdf
[65] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. SPIDER: Enabling Fast Patch Propagation In Related Software Repositories. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1562–1579. doi:10.1109/SP40000.2020.00038
[66] Jahanzaib Malik and Fabrizio Pastore. 2023. An empirical study of vulnerabilities in edge frameworks to support security testing improvement. *Empirical Software Engineering* 28, 4 (2023), 99. doi:10.1007/s10664-023-10330-x
[67] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).
[68] Muskan Mangla. 2023. *Securing CI/CD Pipeline: Automating the detection of misconfigurations and integrating security tools.* Ph. D. Dissertation. Dublin, National College of Ireland.
[69] Steve Mansfield-Devine. 2018. DevOps: finding room for security. *Network security* 2018, 7 (2018), 15–20.

[70] Joel Margolis, Tae Tom Oh, Suyash Jadhav, Young Ho Kim, and Jeong Neyo Kim. 2017. An in-depth analysis of the mirai botnet. In *2017 International Conference on Software Security and Assurance (ICSSA)*. IEEE, 6–12. doi:10.1109/ICSSA.2017.12
[71] Daniel Marjamäki. 2013. Cppcheck: a tool for static c/c++ code analysis. https://cppcheck.sourceforge.io
[72] Jack McBride, Budi Arief, and Julio Hernandez-Castro. 2018. Security Analysis of Contiki IoT Operating System. In *Proceedings of the 2018 International Conference on Embedded Wireless Systems and Networks (EWSN '18)*. 6 pages.
[73] Jonathan Moerman, Sjaak Smetsers, and Marc Schoolderman. 2018. Evaluating the performance of open source static analysis tools. *Bachelor thesis, Radboud University, The Netherlands* 24 (2018).
[74] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Network and Distributed System Security (NDSS) Symposium 2018*.
[75] Timo Müller and Hans-Martin Münch. 2021. Vulnerability digging with CodeQL. https://mogwailabs.de/en/blog/2021/09/vulnerability-digging-with-codeql/.
[76] Nataliia Neshenko, Elias Bou-Harb, Jorge Crichigno, Georges Kaddoum, and Nasir Ghani. 2019. Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-Scale IoT Exploitations. *IEEE Communications Surveys & Tutorials* 21, 3 (2019), 2702–2733. doi:10.1109/COMST.2019.2910750
[77] Anh Nguyen-Duc, Manh Viet Do, Quan Luong Hong, Kiem Nguyen Khac, and Anh Nguyen Quang. 2021. On the Adoption of Static Analysis for Software Security Assessment–A Case Study of an Open-Source e-Government Project. *Computers & Security* 111 (Dec. 2021), 102470. doi:10.1016/j.cose.2021.102470
[78] Eoin O'driscoll and Garret E O'donnell. 2013. Industrial power and energy metering–a state-of-the-art review. *Journal of Cleaner Production* 41 (2013), 53–64.
[79] Open Text. 2023. *Fortify*. https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer
[80] Dipankar Pal. 2023. Deep5050/Cppcheck-Action.
[81] Tao Peng, Wen-Tao Fu, Si-Di Kong, Xiao-Long Li, Ci-Fu Xie, Ting Fu, and Fei Yang. 2024. Application of Continuous Integration in the Development of Embedded Software for Nuclear Power Industry. In *International Symposium on Software Reliability, Industrial Safety, Cyber Security and Physical Protection for Nuclear Power Plant*. Springer.
[82] Quoc-Sang Phan, Kim-Hao Nguyen, and ThanhVu Nguyen. 2023. The Challenges of Shift Left Static Analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.
[83] Dipika Roy Prapti, Abdul Rashid Mohamed Shariff, Hasfalina Che Man, Norulhuda Mohamed Ramli, Thinagaran Perumal, and Mohamed Shariff. 2022. Internet of Things (IoT)-based aquaculture: An overview of IoT application on water quality monitoring. *Reviews in Aquaculture* 14, 2 (2022), 979–992. doi:10.1111/raq.12637
[84] Miro Samek. 2008. *Practical UML statecharts in C/C++: event-driven programming for embedded systems*. CRC Press.
[85] Wedy Freddy Santoso and Dadang Syarif Sihabudin Sahid. 2021. Implementation and performance analysis development security operations (DevSecOps) using static analysis and security testing (SAST). In *Proceeding International Applied Business and Engineering Conference*.
[86] Raphael Satter. 2025. In last-minute reversal, US agency extends support for cyber vulnerability database. *Reuters* (April 2025). https://rb.gy/vuje1j
[87] Ayushi Sharma, Shashank Sharma, Sai Ritvik Tanksalkar, Santiago Torres-Arias, and Aravind Machiry. 2024. Rust for Embedded Systems: Current State and Open Problems. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 2296–2310. doi:10.1145/3658644.3690275
[88] Mingjie Shen. 2023. Pull Request for Improvements to CodeQL. https://github.com/github/codeql/pull/13647.
[89] Mingjie Shen, James C. Davis, and Aravind Machiry. 2023. Towards Automated Identification of Layering Violations in Embedded Applications (WIP). In *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2023)*. 143–147. doi:10.1145/3589610.3596271
[90] Mingjie Shen, Akul Abhilash Pillai, Brian A. Yuan, James C. Davis, and Aravind Machiry. 2025. *Finding 709 Defects in 258 Projects: An Experience Report on Applying CodeQL to Open-Source Embedded Software (Experience Paper)*. doi:10.5281/zenodo.15200316
[91] Mingjie Shen, Akul Abhilash Pillai, Brian A. Yuan, James C. Davis, and Aravind Machiry. 2025. Finding 709 Defects in 258 Projects: An Experience Report on Applying CodeQL to Open-Source Embedded Software (Experience Paper) – Extended Report. arXiv:2310.00205 [cs.SE] https://arxiv.org/abs/2310.00205
[92] NB Soni and Jaideep Saraswat. 2017. A review of IoT devices for traffic management system. In *2017 International Conference on Intelligent Sustainable Systems (ICISS)*. IEEE, 1052–1055.
[93] Nuthan TestMunaiah and Andrew Meneely. 2016. Vulnerability Severity Scoring and Bounties: Why the Disconnect?. In *Proceedings of the 2nd International Workshop on Software Analytics* (Seattle, WA, USA) *(SWAN 2016)*. 8–14. doi:10.1145/2989238.2989239
[94] The Linux Foundation. 2023. Zephyr® Project. https://www.zephyrproject.org/.

[95] The Linux Foundation. 2024. BadgeApp. https://www.bestpractices.dev/en
[96] U.S. General Services Administration. 2023. Application Security Testing. https://rb.gy/35zcyk
[97] Veracode. 2023. *Veracode*. https://www.veracode.com/
[98] Roberto Verdecchia, Emelie Engström, Patricia Lago, Per Runeson, and Qunying Song. 2023. Threats to validity in software engineering research: A critical reflection. *Information and Software Technology* 164 (2023), 107329.
[99] A. J. Vicens and Raphael Satter. 2025. US funding running out for critical cyber vulnerability database, manager says. *Reuters* (April 2025). https://rb.gy/znhdax
[100] Zachary Douglas Wadhams, Clemente Izurieta, and Ann Marie Reinhold. 2024. Barriers to Using Static Application Security Testing (SAST) Tools: A Literature Review. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '24)*. 161–166. doi:10.1145/3691621.3694947
[101] David Wheeler. 2006. Flawfinder. http://www.dwheeler.com/flawfinder.
[102] Elecia White. 2011. *Making Embedded Systems: Design Patterns for Great Software*. O'Reilly Media, Inc.
[103] Wikipedia contributors. 2025. Cumulative distribution function. https://en.wikipedia.org/wiki/Cumulative_distribution_function#Complementary_cumulative_distribution_function_(tail_distribution)
[104] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.
[105] Guest Writer. 2020. The 5 Worst Examples of IoT Hacking and Vulnerabilities in Recorded History. https://www.iotforall.com/5-worst-iot-hacking-vulnerabilities.
[106] Jing Xie, Heather Richter Lipford, and Bill Chu. 2011. Why do programmers make security errors?. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 161–164. doi:10.1109/VLHCC.2011.6070393
[107] Xinyu Zhang, Siddharth Muralee, Sourag Cherupattamoolayil, and Aravind Machiry. 2024. On the Effectiveness of Large Language Models for GitHub Workflows. In *Proceedings of the 19th International Conference on Availability, Reliability and Security* (Vienna, Austria) *(ARES '24)*. Article 32, 14 pages. doi:10.1145/3664476.3664497

Received 2024-10-29; accepted 2025-03-31