

An Empirical Study on the Use of Static Analysis Tools in Open Source Embedded Software

MINGJIE SHEN, Purdue University, USA

AKUL PILLAI, Purdue University, USA

BRIAN A. YUAN, Purdue University, USA

JAMES C. DAVIS, Purdue University, USA

ARAVIND MACHIRY, Purdue University, USA

Embedded software is used in safety-critical systems such as medical devices and autonomous vehicles, where software defects comprising security vulnerabilities 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 security vulnerabilities. One common defense against security vulnerabilities is the use of static analysis, which can offer sound guarantees. While researchers have examined the practices, challenges, and potential benefits of static analysis for many kinds of open-source software, these observations have not been made for EMBOSS. There is little data to guide open-source software engineers and regulators on the cost-benefit tradeoffs of applying (or mandating) static analysis in this context.

This paper performs the first study to understand the prevalence, challenges, and effectiveness of using Static Application Security Testing (SAST) tools on EMBOSS repositories. We collect a corpus of 258 of the most popular EMBOSS projects, representing 13 distinct categories such as real-time operating systems, network stacks, and applications. To understand the current use of SAST tools on EMBOSS, we measured this corpus and surveyed developers. To understand the challenges and effectiveness of using SAST tools on EMBOSS projects, we applied these tools to the projects in our corpus. We report that almost none of these projects (just 3%) use SAST tools beyond those baked into the compiler, and developers give rationales such as ineffectiveness and false positives. In applying SAST tools ourselves, we show that minimal engineering effort and project expertise are needed to apply many tools to a given EMBOSS project. GitHub's CODEQL was the most effective SAST tool – using its built-in security checks we found a total of 540 defects (with a false positive rate of 23%) across the 258 projects, with 399 (74%) likely security vulnerabilities, including in projects maintained by Microsoft, Amazon, and the Apache Foundation. EMBOSS engineers have confirmed 273 (51%) of these defects, mainly by accepting our pull requests. Two CVEs were issued. 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.

1 INTRODUCTION

Our dependence on embedded devices (*e.g.*, IoT devices) and consequently embedded software has significantly increased. Embedded devices control many aspects of our lives, including our homes [17], transportation [14], traffic management [99], and the distribution of vital resources like food [95] and power [89]. The adoption of these devices has seen rapid and extensive growth,

Authors' addresses: Mingjie Shen, Purdue University, West Lafayette, USA, shen497@purdue.edu; Akul Pillai, Purdue University, West Lafayette, USA, pillai23@purdue.edu; Brian A. Yuan, Purdue University, West Lafayette, USA, bayuan@purdue.edu; James C. Davis, Purdue University, West Lafayette, USA, davisjam@purdue.edu; Aravind Machiry, Purdue University, West Lafayette, USA, amachiry@purdue.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2023 Association for Computing Machinery.

XXXX-XXXX/2023/10-ART \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

with an estimated count of over 50 billion devices [13]. Vulnerabilities in Embedded Software (EmS) (enabling these devices) have far-reaching consequences [24, 109] due to the pervasive and interconnected nature of these devices, as exemplified by the infamous Mirai botnet [81].

Open-source Software (OSS) plays an important role in EmS development [21, 45, 74]. Various popular Open-Source Embedded Software (EMBOSS) exist, such as libraries and Real Time Operating Systems (RTOSes) [2]. For instance, FreeRTOS [19] and Zephyr [101], two of the most popular and industry-endorsed RTOSes, are open-source. It is important to ensure that EMBOSS do not contain any vulnerabilities and that suitable vulnerability detection techniques are used to secure them.

The diversity of hardware platforms [86, 108], input mechanisms, and the lack of support for sanitizers [11, 111] make it hard to apply dynamic analysis based vulnerability detection techniques, such as fuzzing [78], to EMBOSS. On the other hand, static analysis vulnerability detection techniques, *i.e.*, Static Application Security Testing (SAST) tools, do not have such requirements. Moreover, the latest State of The Practice (SoTP) tools, such as CODEQL [27] are shown to be effective and can find serious security vulnerabilities in complex codebases [50]. Furthermore, many of these SoTP tools can be easily used in the software engineering pipeline by integrating into Continuous Integration (CI) Workflows, *e.g.*, GitHub Workflows [66]. As we show in Section 3, many critical OSS effectively use SAST tools in their CI Workflows. Furthermore, Chelf *et al.* [37] showed that embedded software can greatly benefit from using SAST tools. However, the use and effectiveness of SAST tools in EMBOSS is unknown.

In this paper, we perform the first systematic study on the use of SAST tools to detect security vulnerabilities in EMBOSS. Specifically, we investigate three research questions:

- **RQ1-Prevalence:** Are SAST tools currently used in EMBOSS?
- **RQ2-Challenges:** Is there any difficulty in configuring SAST tools for EMBOSS?
- **RQ3-Effectiveness:** Can EMBOSS benefit from using SAST tools?

To answer these questions, we curated a corpus of 258 popular EMBOSS projects from GitHub. We also identified a set of 12 SAST tools that can be readily integrated into these projects. We used a combination of automated analysis of CI Workflows and developer surveys to understand the prevalence of SAST tools usage. We used manual analysis and developer surveys to understand the challenges in using SAST tools. Finally, we manually created exemplary SAST CI Workflows for all EMBOSS projects. We used these Workflows to run a modified version of CODEQL (one of the most effective SAST tools) on each project and analyzed the corresponding results.

To summarize our results: available SAST tools — specifically, CODEQL — are easy to configure and substantially outperform EMBOSS developers’ common practice, which is the compiler’s warnings. We identified that only 10 (4%) projects use SAST tools as part of their CI Workflows. Furthermore, our developer survey indicates that despite developers being aware of SAST tools, most developers do not use them on EMBOSS projects. Most developers claim to use strict compiler warnings (*i.e.*, `-Wall`, `-Wextra` and `-Werror`), but they are less effective compared to SoTP SAST tools, as we show in Section 7.4. We identified that CODEQL, one of the most effective SoTP SAST tools, cannot handle diverse build systems of EMBOSS repositories and consequently fails to run on many of them. However, we were able to fix this with minimal engineering effort and created CI Workflows enabling the execution of CODEQL on EMBOSS repositories. *We found a total of 540 defects, with 399 (74%) being security vulnerabilities, demonstrating the need to use SAST tools on EMBOSS projects.*

Our contributions are:

- We performed the first study to understand the prevalence and benefit of using SAST tools in EMBOSS. We observed that current techniques used by developers (compiler warnings) are ineffective compared to a SoTP SAST tool.

- As part of our study, we curated a list of 258 EMBOSS projects and created exemplary GitHub Workflows – encapsulating all the necessary compilation steps – enabling execution of SAST tools. This is the first large-scale embedded software dataset with the necessary compilation infrastructure.
- We executed CODEQL on our EMBOSS dataset using the created Workflows. We identified a total of 540 defects (399 (74%) security vulnerabilities) across all projects, including projects maintained by reputed groups such as Apache, Microsoft, and Amazon. We have reported all these defects and raised pull requests. The developers have already confirmed 273 (51%) of these defects, mainly by accepting our pull requests.
- We open-source all our datasets and GitHub Workflows, enabling future science.

Significance for software engineering: Empirical software engineering research has a substantial body of knowledge on open-source software, but has focused on IT or general-purpose software [23]. We present a large-scale evaluation of embedded open-source software, reporting on both the effectiveness of SAST tools and on developers’ perceptions. Across 258 EMBOSS, the CODEQL SAST tool finds hundreds of defects with modest per-repository configuration and a low false positive rate. EMBOSS software developers can use this tool to easily improve software quality.

2 BACKGROUND

2.1 Embedded Software, RTOSes, and Open-Source

Embedded software is designed to run on embedded systems, ranging from industrial controllers [30] to resource-constrained microcontroller-based IoT devices [17]. As mentioned in Section 1, these devices control various aspects of our daily lives. Unlike regular computers, embedded devices use specialized Real Time Operating Systems (RTOSes) designed for reduced-resource environments (e.g., real-time scheduling, low power consumption, low memory overhead). There are 31 different RTOSes [2], with the majority (26) of them being open-source and developed in unsafe languages such as C/C++. Examples of RTOSes include RIOT, Contiki, FreeRTOS, and Azure RTOS.

Open-source Software (OSS) is an essential part of the software supply chain of embedded systems. The inherent advantages of open-source software, such as long-term sustainability and accessibility to source code for debugging purposes, have been acknowledged and appreciated in the embedded software industry [73, 74]. Most of Open-Source Embedded Software (EMBOSS) is in C/C++, and studies [15, 16] show that many EMBOSS use many unsafe statements.

2.2 SAST Tools

Embedded software, with its hardware-coupled nature [86] and the lack of necessary emulation support [44], makes it hard to use dynamic analysis techniques in a scalable manner. Static Application Security Testing (SAST) tools are specially designed static analysis techniques to find security vulnerabilities effectively. They do not require execution support, making them attractive to use on embedded software. Furthermore, as we show in Table 2, many SoTP SAST tools have the necessary plugins to be easily integrated into CI pipelines.

2.2.1 Availability. 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 [32], 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` [82] and `flawfinder` [106], which can be readily used but do not provide any guarantees. Several works [35, 49, 72, 85] 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 state-of-the-art and allows developers to customize the tool to reduce false positives [61]. Other notable tools include Fortify [90], Checkmarx [36], and Veracode [105].

2.2.2 CODEQL. This is a recent SAST tool created and 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 [27]. 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 codebases, including the Linux kernel [4, 5, 50].

2.3 Continuous Integration (CI) Workflows

Continuous Integration (CI) pipelines or Workflows [60] have become ubiquitous in the software development lifecycle. They automate various software development processes, such as building, testing, and deploying code. The GitHub CI with its close integration with GitHub infrastructure, is the most popular CI framework for the projects hosted on GitHub [58]. GitHub CI supports *Actions*, *i.e.*, modules or plugins that enable easy development of Workflows. More than 19K actions are available on GitHub Marketplace [56]. For instance, one can use `actions/cmake-action` Action [83] to build a `cmake` project.

Several works [29, 79, 80] show that CI Workflows provide a perfect place to run SAST tools as they can be easily integrated into the development pipeline. Furthermore, there are several GitHub Actions (Table 2) that enable running various SAST tools, including CODEQL, as part of a GitHub Workflow.

3 MOTIVATION

Most embedded software are developed in unsafe languages, *i.e.*, C/C++. Many works [62, 87, 93, 97] emphasize the importance of using SAST tools on software projects, especially those using unsafe languages such as C/C++. Many security and government organizations [1, 6] also recommend the use of SAST tools. Many software engineering tasks are being automated in CI pipelines, which provide an ideal place to use SAST tools. Our analysis shows that 958 (19%) of top 5K critical and extremely critical OSS projects (according to OSSF criticality score explained in Section 4.1.3) on GitHub use one or more SAST tools as part of their CI pipelines, *i.e.*, GitHub Workflows.

It is important to ensure that SAST tools are also used in EMBOSS. Many effective SAST tools require compilation of the underlying software and assume certain coding idioms (*e.g.*, use of standard libraries). These tools are often evaluated [35, 49, 72, 85] on traditional or non-embedded software. However, embedded software differs [108] from traditional software in design, library usage, organization, build system, and toolchains. *It is unclear how challenging it is to use existing SAST tools and their effectiveness on EMBOSS.* A decade ago, Torri *et al.* [102] surveyed ten different free/open-source SAST tools and evaluated their use on five embedded applications. Their results found that the tools present widely different results, and most are not ready to be applied to embedded systems. Notably, numerous modifications were required to the source code and needed to deal with diverse build processes. However, *no work exists to understand the prevalence and effectiveness of SAST tools in EMBOSS at scale.*

4 STUDY METHODOLOGY

Our study addresses three research questions:

- **RQ1-Prevalence (Section 5):** Are SAST tools currently used in EMBOSS?
- **RQ2-Challenges (Section 6):** Is there any difficulty in configuring SAST for EMBOSS?

- **RQ3-Effectiveness (Section 7):** Can EMBOSS benefit from using SAST tools?

We use mixed-methods study, systematic analysis, and large-scale evaluation to investigate these questions. We start with details of our data collection methodology and tool selection. Then, we present the methodology, results, and analysis for each research question in turn.

4.1 Embedded Software Dataset

We aim to collect a set of representative and popular EMBOSS. Embedded systems are usually powered by an RTOS, which provides the necessary library and scheduling support for various application components. To collect a representative EMBOSS dataset, we use a two-pronged approach as shown in Figure 1.

4.1.1 GitHub Crawling. We searched (on Apr 8, 2023) 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. We sorted the resulting repositories according to their popularity (*i.e.*, number of stars) and collected the top 250. Also, all these repositories have more than 100 stars, indicating their popularity. We manually checked each repository to filter false positives by removing non-embedded repositories. For instance, we filtered out a machine learning project that also contained the word “embedded” in its keywords. In total, this resulted in 238 repositories.

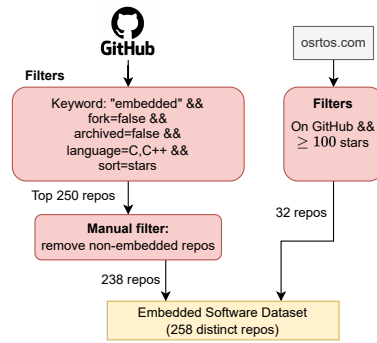


Fig. 1. Our two-pronged approach to collecting embedded software dataset.

4.1.2 Well-Known Sources. We collected RTOSes from osrtos.com [2], which maintains the list of all open-source RTOSes released to date. Specifically, we selected those available on GitHub and with more than 100 stars. This resulted in a total of 32 repositories.

4.1.3 Summary. We combined the repositories and de-duplicated them, resulting in a total of 258 unique EMBOSS repositories. Table 1 shows the summary of all repositories along with their fine-grained categorization (performed manually). Appendix E in our extended report [26] shows Source Lines of Code (SLOC) statistics of these repositories. Most repositories are reasonably large, with a median ranging from 10K - 100K SLOC, which agrees with numbers published in other studies [98].

Measuring Importance of Software in the Dataset Using the OSSF Criticality Score The Open Source Security Foundation (OSSF) organization [46] created a mechanism to compute a criticality score for GitHub repositories. This score is not commonly used in the empirical software engineering literature, but it is receiving attention in practice [40, 75] because it is useful for security analysts scanning large datasets to triage the security vulnerabilities. Since SAST tools are often used for this purpose, we define the OSSF criticality score here.

The OSSF criticality score [25] of a repository is a number ranging from 0.0 - 1.0 intended to measure the importance of a repository. The score is computed through a formula that incorporates measures of popularity, dependents, activity, and other attributes. A score s from $0.0 \leq s < 0.2$, $0.2 \leq s < 0.4$, $0.4 \leq s < 0.6$, $0.6 \leq s < 0.9$, $0.9 \leq s$ indicates low, medium, high, critical, extremely critical severities, respectively. For instance, The Swift language frontend (with 2.4K stars) [10] and contiki-os (an RTOS with 3.6K stars) [7] have criticality scores of 0.51, indicating high severity projects. The Linux kernel (with 157K stars) [103] has a criticality score of 0.88, indicating a critical

project. The Node.js runtime has (with 97.6K stars) [8] has a score of 0.99, indicating an extremely critical project.

In our EMBOSS dataset, most of the 13 categories have projects with a median criticality score ranging from 0.4-0.5, indicating projects of considerable importance. The box plot of OSSF criticality score (Section 4.1.3) for EMBOSS in each category is in Appendix E of our extended report [26].

Table 1. Categorization of repositories in our EMBOSS dataset. *Total*: Medians across corpus, not by category.

Category	# Repos	Example Repo	Median GH stars	Median SLOC	Median Crit. Score
Hardware access library (HAL)	18	grbl	303.5	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	json library, nanopd	313.5	2,547	0.41
Language support (LS)	33	micropython	479	33,389	0.42
UI utilities (UI)	14	flutterpi	584.5	56,712	0.46
Embedded applications (APP)	32	Infinitime	508	22,662.5	0.39
OSes (OS)	42	FreeRTOS, Zephyr	727.5	409,667.5	0.47
Memory Management Library (MML)	4	tinyobjloader-c	242.5	6,205.5	0.34
Other General Purpose Library for Embedded Use (GPL)	22	tinyprintf	391	12,742.5	0.35
Other (OT)	6		368.5	94,805	0.43
Total	258		406.5	33545	0.43

4.2 Usable SAST

Our goal is to find SAST tools that can be readily used on the collected GitHub repositories. Given that GitHub Actions are expected to be stable, easy to use, and can be seamlessly integrated into repositories, we used GitHub Marketplace and found GitHub Actions designed for SAST purposes.¹ We manually filtered out pre-release Actions due to their instability and/or lack of documentation. There were 6 commercial SAST tools, which we omitted as they require purchases of licenses or subscriptions and place restrictions on scientific publications. Furthermore, for individuals, small teams, or organizations with limited financial resources, the cost of commercial tools may be prohibitive, making them less feasible compared to free or open-source alternatives.

4.2.1 Summary. This resulted in a total of 12 GitHub Actions using various SAST tools as shown in Table 2. Nine of these Actions are plug-and-play, meaning they do not need any repository-specific configuration. In other words, the steps to use the Action do not vary with the underlying repository. For instance, the CODEQL Action (*i.e.*, `github/codeql-action` [52]) is a plug-and-play Action because every repository uses the same steps to use the Action, whereas the `Frama-C/` `github-action-eva-sarif` Action [47] requires developers to create a special Frama-C Makefile and provide the path to it.

Effectiveness on C/C++ Juliet Test Suite [31]: Recent work [20] shows developers prefer plug-and-play SAST tools as they do not need project-specific configuration. To measure baseline effectiveness, we tested all plug-and-play tools on the Juliet Test Suite, a labeled dataset commonly used to test SAST tools [88].

The last column of Table 2 shows the results. Except for CODEQL, all other tools either failed or did not finish within 6 hours, which exceeds the maximum time allowed for a job by many CI platforms, such as GitHub CI [57]. CODEQL took 40 minutes. The reasons for failure include 1)

¹The query is `category=security&type=actions&query="C C++"` and `category=code-quality&type=actions&query="C C++"`.

the Static Analysis Results Interchange Format (SARIF) file produced by the tool is invalid, and 2) the argument list given to the SAST tool is too long. We checked the results of CODEQL and found that it raised 11,101 warnings with a precision of 71% (7,904/11,101).²

Table 2. List of “usable” SAST GitHub Actions. These are GitHub Actions that perform SAST on C/C++ repositories, not including pre-release or commercial tools.

Name of GitHub Action	Plug-and-play?	Underlying tool(s)	Juliet Test Suite results
From Well-established Organizations			
github/codeql-action [52]	Yes	CODEQL	7,904 true positives
cpp-linter/cpp-linter-action [38]	Yes	clang-format, clang-tidy	Not finished in 6 hours
trunk-io/trunk-action [104]	No (Bazel/CMake projects required)	clang-format, clang-tidy, include-what-you-use, pragma-once	N/A
Frama-C/github-action-eva-sarif [47]	No (Frama-C Makefile required)	Frama-C	N/A
From Independent Developers			
IvanKuchin/SAST [63]			
deep5050/flawfinder-action [91]	Yes	flawfinder	Error
david-a-wheeler/flawfinder [106]			
Syndelis/cpp-linter-cached-action [68]	Yes	clang-format, clang-tidy	N/A
deep5050/cppcheck-action [92]	Yes	cppcheck	Not finished in 6 hours
Konstantin343/cppcheck-annotation-action [67]	Yes	cppcheck, clang-tidy	Error
JacobDomagala/StaticAnalysis [42]	No	clang	N/A
whisperity/codechecker-analysis-action [107]	No (Compilation DB required)	clang	N/A

5 RQ1: PREVALENCE OF SAST TOOLS

5.1 Method

In this research question, we plan to understand the prevalence of SAST tool usage in EMBOSS repositories. In general, it is impossible to automatically identify whether an EMBOSS project uses SAST tools because developers might use them out-of-band with no hints/indications in the corresponding repository. *e.g.*, developer might manually run a SAST tool before every release. We perform a mixed-methods study by Workflow analysis and developer surveys to investigate this research question.

5.2 Workflow Analysis

We noticed that 42% (109/258) of the EMBOSS repositories use GitHub Workflows to automate building and testing the underlying codebase. As mentioned in Section 2.3, these Workflows are an ideal place to use SAST tools. We performed an automated analysis of Workflows in each repository to detect the usage of SAST tools. Specifically, for each Action used in a Workflow, we check if it is a SAST tool by checking its category (Section 4.2). Next, we manually check every matching Action to validate that it is indeed a SAST tool. We found that 248 of the repositories *do not use* any SAST tool, 10 of them use free SAST tools, specifically, CODEQL, and none of them use commercial SAST tools. Upon further analysis, we found that of the 10 repositories that use CODEQL, 7 are mis-configured (use a deprecated Action version) while the other 3 configure it correctly.

Given the large number of repositories (248) and complexity of Workflows, it is impractical to manually verify all the negative results, *i.e.*, Workflows missing SAST tools. Instead, we performed a random sampling of 20 Workflows and manually checked them. We found only two false negatives, *i.e.*, 10% false negative rate. Specifically, RIOT-OS/RIOT runs its own static test tools in a

²We count a reported flaw as a true positive if the reported location matches that of a ground truth bug.

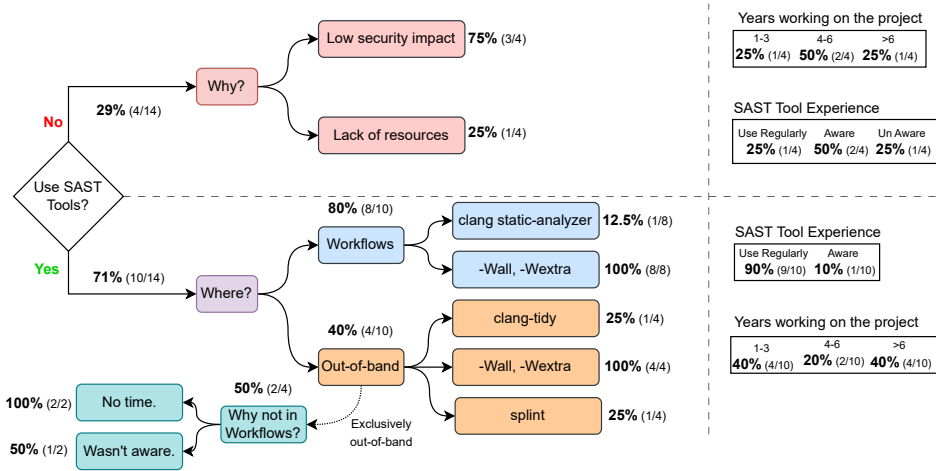


Fig. 2. Summary of our developer survey on the use of SAST tools.

Docker container through Workflows and `InfiniTimeOrg/InfiniTime` runs `clang-tidy` in a script. Our automatic analysis failed to detect them because we did not parse the contents of the scripts used in Workflows.

5.3 Developer Survey

We did an anonymous online survey with repositories' maintainers to identify out-of-band usage of SAST tools. First, we identified all the repositories for which we did not find any SAST tool usage in their Workflows. For each of these repositories, we collected emails (from the public GitHub profile) of users who merged recent pull requests and/or made commits directly to the repository and sent them an email with the link to our survey.³ We were able to find the maintainers' email only for 104 (out of 258) projects. Our study is approved by the Institutional Review Board (IRB) under #2023-1062. The exact questions of the survey are listed in Appendix F of our extended report [26]. We got responses for 14 projects (a 13% response rate, comparable with other works that survey developers from GitHub). Figure 2 shows the summary of responses.

Use of SAST Tools. 71% (10/14) of the projects claim to use SAST tools, out of which 80% (8/10) claim to be using them as part of their GitHub Workflows. This shows that developers are aware of the possibility of using SAST tools in GitHub Workflows. However, only one project uses an explicit SAST tool (*i.e.*, `clang static-analyzer`). Our automated analysis (Section 5.2) did not find this as the tool might be used as part of a script (*e.g.*, `./test.sh`). Since our analysis does not consider the script, we failed to detect this. Many projects claim to use stringent compiler warnings (*i.e.*, `-Wall`, `-Wextra`) instead of explicit SAST tools. Our automated analysis did not check for these flags, which could be used as part of the build scripts. As we show in Section 7, stringent compiler warnings are ineffective at detecting security vulnerabilities.

Only 40% (4/10) projects claim to use SAST tools out-of-band (*i.e.*, outside of Workflows). Only 2 of those 4 projects use SAST tools exclusively out-of-band (*i.e.*, undetectable by looking at Workflows). This suggests that analyzing Workflows is effective to identify the use of SAST tools.

³This is consistent with GitHub's Acceptable Use policy [55]: profile emails are public information and the number of emails sent was small enough to not qualify as mass spam.

Not Using SAST Tools. 29% (4/14) of the projects do not use any SAST tools. Most of these engineers (3 of the 4 projects) believe the security vulnerabilities in the corresponding projects have a low impact. However, these projects have OSSF criticality scores of ~ 0.3 (medium), ~ 0.4 (high), and ~ 0.6 (critical), respectively. These developers may underestimate the severity of security issues, confirming previous studies [71, 110]. The developers of another project reported insufficient resources (*e.g.*, time). Unfortunately, this project is one of the most popular (7.3K stars) open-source C++ library suites for embedded systems and is used in many critical embedded projects with OSSF score of 0.67 (a critical project).

Comments on Using SAST Tools. A few (5) developers mentioned that the effectiveness of SAST tools on embedded software is questionable and might result in many false positives.

5.4 Results

Our methods show that most of EMBOSS repositories do not use SAST tools. Specifically, our Workflow analysis show that only 10 (4%) repositories use SAST tools as part of their CI Workflows. Merely 3 (1%) repositories correctly use SAST tools, specifically CODEQL, a small number compared to non-embedded repositories, where many similarly-popular repositories (958 of ~ 5000 or 19%) correctly use it (Section 3). Our developer survey also revealed similar results, wherein most repositories do not use any explicit SAST tool. Furthermore, developers mentioned using strict compiler warnings.

Finding 1: Most (97%) of the EMBOSS repositories *do not use SAST tools*.

Finding 2: Our survey indicates that many EMBOSS repositories rely on compiler warnings instead of dedicated SAST tools.

Finding 3: Our survey shows that most developers are aware of CI Workflows and use them to run their SAST tools.

6 RQ2: CHALLENGES IN EFFECTIVELY USING SAST TOOLS

In this research question, we want to know how challenging it is to effectively use SAST tools on EMBOSS repositories. We focus on SoTP tools as listed in Table 2. In particular, we focus on plug-and-play tools, which have a fixed set of steps for any repository.

6.1 Method

For each SAST tool, we picked the most popular Action implementing it. For instance, for `flawfinder`, we picked `david-a-wheeler/flawfinder` — this repository has the most stars among the Actions offering this tool. Table 3 shows Actions selected for each SAST tool. For each selected Action, we reviewed the documentation in order to create a Workflow for the Action. We created a Workflow for each Action and executed it on each repository.

6.2 Results

Table 3 shows the results of this experiment.

Table 3. Results of SAST tools on EMBOSS repositories.

Action	Result format	# Success Repo	# Failure Repo	Reasons for failure	Total # warn	Median # warn	Precision
<code>david-a-wheeler/flawfinder</code>	SARIF	176	82	Invalid SARIF, Python Error	4,637	12	20% (64/316)
<code>cpp-linter/cpp-linter-action</code>	GCC error msg	230	28	Timeout, Python Error	212,228	111	0% (0/213)
<code>deep5050/cpcheck-action</code>	GCC error msg	256	2	Timeout	31,873	19	58% (116/200)
<code>CodeQL Autobuild</code>	SARIF	74	184	Autobuild failure	471	0	96% (154/160)

6.2.1 Tool Execution. The encapsulation of all the steps to set up (e.g., dependency installing) a tool in GitHub Action makes it straightforward to run each tool on all the repositories in the EMBOSS dataset. More concretely, it requires placing the tool Workflow file in `.github/workflows` folder of the repository and pressing a button on the repository webpage on GitHub.

6.2.2 Tools Failures. Except for CODEQL, all Actions ran successfully on most of EMBOSS repositories. Nonetheless, there are many repositories where Actions failed, and Table 3 also shows the common reasons for the failure. Despite the success of CODEQL on Juliet Test Suite (Table 2), CODEQL failed on the majority of EMBOSS repositories. The reason for this is the failure of the `autobuild` step. This mechanism is good at handling non-embedded projects that use standard build procedures, such as `make`, `cmake`, etc. However, embedded projects usually use a customized build process with diverse toolchains. For example, `ARMmbed/DAPLink` uses a customized Python script to build the project. Such custom build processes cannot be handled by `autobuild`.

6.2.3 Tools Effectiveness. As shown in Table 3, `flawfinder` and CODEQL produce results in SARIF format, which makes it easy to navigate to the source location corresponding to a warning. However, the other tools, i.e., `cpp-linter`, and `cppcheck`, produce a non-standard textual message, making triaging warnings tedious. This is in line with the observation made by a recent study [20], where developers were concerned about ineffective error reporting mechanisms of SAST tools.

Table 3 also shows the total and median number of warnings across all repositories on which the corresponding tool ran successfully. At a high level, except for `cpp-linter` (with 111), all other tools have a moderate number (< 20) of median warnings. The median warnings are 0 for CODEQL because it did not produce warnings on more than 50% of repositories.

The lack of ground truth data on our EMBOSS repositories makes it challenging to evaluate the precision/recall of each tool. Given the large number of warnings, it is infeasible to check them all. Therefore, we randomly sampled warnings to determine if they were legitimate. Specifically, for each tool, we randomly selected 30 repositories with fewer than 20 warnings and manually checked each warning for these repositories. The last column in Table 3 shows the precision. CODEQL has the highest precision of 96% — this is unsurprising given its effectiveness on Juliet Test Suite (Table 2). At the other end of the spectrum, `cpp-linter` had the least precision, 0%! Most of `cpp-linter`'s warnings were related to missing header files, pre-processor directives, and other compile-time errors. `cpp-linter` uses a fixed mechanism for compilation and cannot handle custom build setup in each repository, resulting in false positives.

Finding 4: Few SAST tools produce warnings in a non-standard text format, hindering their usability. This aligns with a recent study [20], where developers raised this as a concern for not using SAST tools.

Finding 5: Except for CODEQL, all other SAST tools executed successfully on the majority of EMBOSS repositories. CODEQL tries to compile the target repository automatically but fails to handle the diverse build infrastructure of the majority (184 (71%)) repositories.

Finding 6: Our preliminary evaluation based on random sampling shows that CODEQL has the highest precision on EMBOSS repositories.

7 RQ3: EFFECTIVENESS OF SAST TOOLS ON EMBOSS

We want to understand if EMBOSS can benefit from using an effective SAST tool. Specifically, we want to validate the developers' perspective that SAST tools may not be effective on embedded software (Section 5.3). In Section 6, we ran SoTP SAST tools on EMBOSS repositories. However,

given the large number of warnings, it is infeasible to verify all of them manually. Therefore, in this section we focus on the best tool identified so far.

7.1 Method

We select the most effective SAST tool among those we considered so far. We configure it for each repository in our corpus. Then, we manually analyze all the alerts and warnings to understand its effectiveness specifically on EMBOSS.

7.1.1 Tool Selection: CODEQL. We picked CODEQL for our experiment as it satisfies all our requirements. First, it is the most effective open-source tool for bug finding based on a previous study [70] and also based on our evaluation of CODEQL on the Juliet Test Suite (Table 2) and random sampling (Section 6.2.3). Second, it is maintained by Microsoft and has an active and responsive community. Finally, several reports [4, 5, 50] show that CODEQL was able to find several high-impact security vulnerabilities in large and well-tested codebases. Furthermore, CODEQL’s extensive documentation [53], automated scanning [51], and various other support tools [43] make it one of the easiest tools to integrate in CI Workflows. We used CODEQL command-line toolchain release 2.13.1 (May 3, 2023) and the query repository based on the commit 202037e925 (May 12, 2023).

7.1.2 Build Scripts Creation. As described in Section 6.2.3, CODEQL failed to compile various (184) EMBOSS repositories because of using non-standard build setups. We tried to manually create build scripts for these repositories by referring to their documentation and CI scripts. We also made the build scripts cover as much part of the codebase as possible (e.g., by compiling all example applications and all supported architecture and boards whenever possible). Even for the repositories where CODEQL’s `autobuild` worked, we manually created build scripts to cover most of the codebase. *We were able to successfully create build scripts for 154 (60%) repositories.* For the other 104 repositories, the build instructions were either missing (17), too complex (i.e., unavailable toolchains or dependencies) (51), or did not work (36). *On average, it took 45-60 minutes to create self-contained build scripts for each repository.*

7.1.3 CONFIGURING CODEQL. CODEQL supports many suites (i.e., collections of queries). We choose `cpp-security-and-quality` as it contains the most queries – 166 in total. However, we excluded a few of its queries and modified a few others to improve their precision.

Excluded Queries: We omitted 9 queries from the CODEQL suite for three reasons: (1) they identify code smells but not necessarily defects; (2) the potential risk of the corresponding defects is relatively low (e.g., converting the result of an integer multiplication to a larger type is an issue only when the result is too large to fit in the smaller type); or (3) they are not applicable to embedded software. Appendix G in our extended report [26] shows the complete list of excluded queries and our detailed rationale.

Modified Queries: We modified 3 queries to improve their precision and ignore certain restrictions. Appendix H in our extended report [26] describes all 3 changes. For example, we modified `cpp/constant-comparison` to only report comparison that is always false because we found that always-true comparison is usually not a defect. 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)`. CODEQL accepted one of our query modifications into their main repository [3].

7.1.4 Workflow Creation. We created GitHub Workflows for all the successfully built repositories (154). These Workflows, when triggered, invoke the necessary build scripts and run CODEQL with the required configuration.

7.2 Results

The scrutiny of the results demanded the dedicated efforts of two authors spanning 25 days. We analyzed the results of 143 (out of 154) repositories. The remaining repositories have a substantial number of defects, and we did not have sufficient time to analyze them thoroughly. In total, across all the 143 analyzed repositories CODEQL reported 578 errors (potential defects) and 2,294 warnings (code-smells, undefined behaviors, etc.). Table 4 summarizes the results.

7.2.1 Defects Discovered. We discovered 540 defects (spanning 83 repositories), of which 399 are security vulnerabilities (spanning 71 repositories). Note that multiple errors or warnings might be raised for a single defect. Figure 3 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 (344 (64%)). The Memory management libraries with the least number (4) of repositories also have the least defects (1). Also, the number of defects is proportional to the size of the codebase. For instance, although NET has more repositories than OS, *i.e.*, The median SLOC of OS, *i.e.*, 409K, is higher than NET, *i.e.*, 36K. Consequently, the number of defects in OS repositories is larger than NET, *i.e.*, 166 v/s 96. *In summary, the density of defects remains constant across various EMBOSS repositories.*

Defects in Each Repository. Figure 4 shows CDFs of the number of all defects and security defects in each repository. A point (x, y) on a line indicates that $y\%$ of repositories contain less than x corresponding type of defects. The left-most point on both the lines indicates that there are 60% (83) repositories with at least one defect, and 51% (71) repositories with at least one security defect. The security defects line has almost the same trend as total defects, indicating that most defects in all repositories are security-relevant. Although 90% of the repositories have less than ten total defects, there are still a considerable number (8) of repositories with a large number of defects. Table 5 shows the top 5 repositories with the highest total defects, security defects, and their OSSF criticality score.

Common Types of Security Defects. We found several classes of security defects across all repositories. Figure 6 shows the top 10 major types of security defects (*i.e.*, vulnerabilities) [54] found along with the corresponding number of defects. We will discuss the top three types of defects and corresponding rules:

Table 4. Summary of CODEQL results and their analysis.

Number of ...	Value
Setup	
Repos in dataset	258
Repos built	154
Repos analyzed	143
CODEQL Results	
Errors reported	578
Warnings reported	2,294
Manual Analysis	
Defects discovered	540
Repos where defects were discovered	83 (60%)
Security defects discovered	399
Repos where security defects were discovered	71 (51%)
Responsible Disclosure	
Defects confirmed	273
Security defects confirmed	219
Pull requests raised	139
Pull requests merged	81
CVEs issued	2

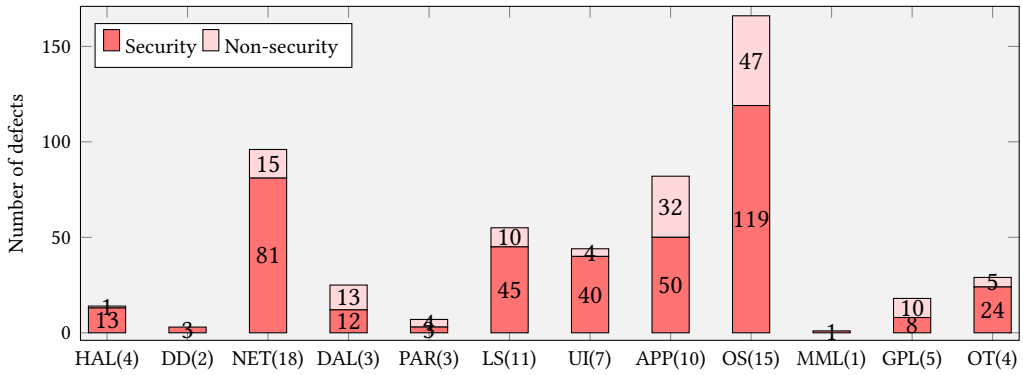


Fig. 3. Number of defects of each type in EMBOSS of various categories (Table 1). The number next to the category indicates the number of repositories containing at least one defect.

Table 5. EMBOSS repositories with the top 5 highest total defects and security defects.

Total Defects			Security Defects		
Repo	Criticality Score	Num	Repo	Criticality Score	Num
contiki-ng/contiki-ng	0.67	32	openlgtv/epk2extract	0.45	27
openlgtv/epk2extract	0.45	29	gozfree/gear-lib	0.43	24
ARMmbed/mbed-os	0.72	27	raysan5/raylib	0.70	23
introlab/odas	0.46	25	contiki-ng/contiki-ng	0.67	22
gozfree/gear-lib	0.43	24	jnz/q3vm	0.34	18

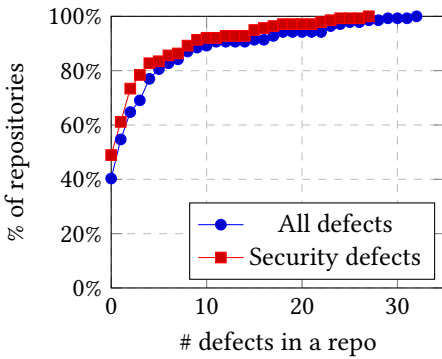


Fig. 4. CDFs of # of all and security-relevant defects in a repository.

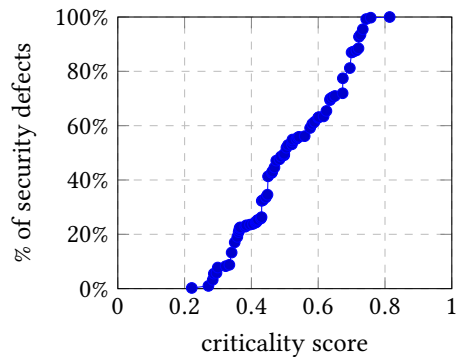


Fig. 5. CDF of the severity of security defects

- `cpp/inconsistent-null-check`: This rule identifies cases in which a function return value is not checked for `NULL`, while most other calls to the same function check the return for `NULL`. Developers should always check the return value of such function if it may return `NULL` to prevent subsequent null pointer dereference. This rule detected 122 such instances. Appendix A in our extended report [26] shows an instance of this issue from the `ARMmbed/mbed-os` repository.

```
// 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 1. The memory pointed by `priv` can be freed (▲) inside the `if` condition but will be accessed later outside, resulting in use-after-free (●).

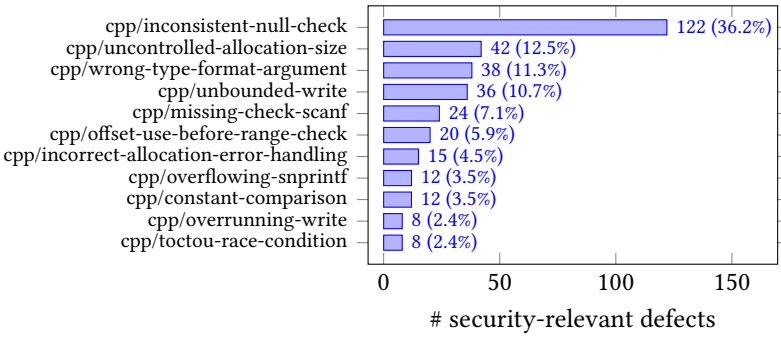


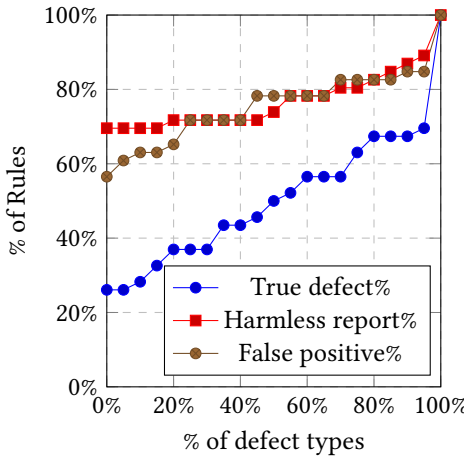
Fig. 6. Top 10 CodeQL queries by # of security-relevant defects found. (Counts and Percentages)

- `cpp/uncontrolled-allocation-size`: This indicates cases where the allocation size argument of a memory allocation call (e.g., `malloc`) is a multiplication of operands derived from potentially untrusted input (e.g., user input). When the operands hold a really large value, an integer overflow [41] might occur and yield a significantly smaller value than intended. Hence, the size of the allocated memory may be considerably less than expected. Subsequent attempts to access the allocated buffer would lead to buffer overflows. This rule detected 42 instances. Appendix A in our extended report [26] shows an instance of this defect in the `embox/embox` repository.
- `cpp/unbounded-write`: This rule detects the class out-of-bound write vulnerabilities [94]. Specifically, this includes analysis of potentially dangerous function calls (e.g., `strcpy`, `scanf`) to check whether these are used properly with valid arguments. This rule detected 36 vulnerabilities of potential buffer overflow. Appendix A in our extended report [26] shows an instance of this vulnerability in the `aws/aws-iot-device-sdk-embedded-C` repository.

Severity of Security Defects. The severity of a security bug depends on its exploitability and the criticality of the underlying software [39, 100]. Given the large number of defects, manually assessing exploitability is intractable. Instead, we use the OSSF criticality score (Section 4.1.3) of the target repository to assess the severity of a bug. Figure 5 shows the CDF of the severity of security defects. Specifically, a point (x, y) on the line indicates $y\%$ of the defects have severity less than or equal to x . 50% (~ 100) bugs have a severity score of more than 0.5, which represents high-severity repositories (Section 4.1.3). Specifically, 40% of bugs have a score of more than 0.6, representing vulnerabilities in critical repositories. For instance, we found an off-by-one error in `micropython/micropython` (Listing 2) and a use-after-free in `apache/nuttx` (Listing 1), a RTOS with a score of 0.69 – both of these are critical projects.

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 [69]. Similarly, the rule `cpp/virtual-call-in-constructor` detects calls to virtual functions in a constructor. They may not resolve to the intended function [33]. We provide more details about the major types of *non-security* defects in Appendix C of our extended report [26].

7.2.2 Precision. To study the precision of CODEQL results, we sampled 37 repositories and manually categorized all errors and warnings 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. True positives are further classified as true defects and harmless reports. A harmless report means that although the result matches the rule’s intention, it is not a defect in the specific context of the code. *e.g.*, `cpp/inconsistent-null-check` might report a function call with a missing `NULL` check of its return value. Although the warning is true, it might be infeasible in the specific context. The overall percentages of true and false positives are 77% (362/468) and 23% (106/468), respectively. Out of 362 true positives, there were 158 harmless reports. Figure 7 shows the CDF of the percentage of rules and their contribution to different types of results. Specifically, a point (x, y) on a line indicates $y\%$ of the rules have resulted in less than or equal to $x\%$ of the corresponding results. Approximately 60% rules had no false positives, and 22% had no true positives. This indicates that false positives are polarized, and a few rules contribute to the majority of false positives. We provide a detailed discussion on rules contributing to the false positives in Appendix B of our extended report [26].



```
// micropython/extmod/vfs_lfsx.c
size_t from = 1;
char *cwd = vstr_str(&self->cur_dir);
while (from < CWD_LEN) {
    for (; cwd[from] == '/' && from < CWD_LEN;
        ↪ ++from) {
        // Scan for the start
    }
    ...
}
```

Listing 2. The offset `from` is used before the range check (▲), leading to an off-by-one error (●).

Fig. 7. CDFs of the percentages of rules and corresponding defect types.

7.3 Responsible Disclosure and Developer Response

We responsibly disclosed all the identified defects either by raising issues or pull requests with appropriate patches (where possible). The bottom part of Table 4 shows the summary of our responsible disclosure. In total, 51% (273/540) 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

aws/aws-iot-device-sdk-embedded-C 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.” We are in the process of raising pull requests to integrate our CODEQL Workflows into corresponding repositories.

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 [48, 59].

7.3.1 CVE Assignment. Our research confirmed the observation of prior work [76], that security defects are often fixed “silently”, without tracking via a Common Vulnerability Enumeration (CVE). 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: `mbedt1s` issued CVE-2023-BLINDED, and `contiki-ng` issued CVE-2023-BLINDED. 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.

7.4 Effectiveness of Stringent Compiler Flags

As reported in Section 5.3, EMBOSS developers often use strict compiler flags/warnings instead of SAST tools. We evaluated the effectiveness of these flags in finding the defects detected by CODEQL. We used security bug test case files from the CODEQL repository for this experiment. These are simple test cases (< 10 lines), each containing an obvious security issue, e.g., passing an invalid pointer types to a function call. We selected test cases to cover all 82 of the identified defect types and compiled them using the latest version of gcc, i.e., 11.4.0, with strict warnings (`-Wall, -Wextra -Werror`). This configuration of gcc found issues in only 17 (21%) defect types (details in our extended report [26]). gcc was able to find certain simple security issues, such as direct use of `strcpy`. However, it did not find more complex ones related to code quality, such as inconsistent null check. *Our results indicate that the current EMBOSS practice of reliance on gcc warnings is inadequate.*

Finding 7: The default build step of CODEQL (`autobuild`) fails on many EMBOSS projects. However, getting CODEQL running takes minimal engineering effort, 45-60 minutes per project.

Finding 8: CODEQL discovers many security and non-security defects in EMBOSS repositories, including repositories maintained by reputed organizations like Amazon and Microsoft.

Finding 9: The strict compiler warnings commonly used by developers (instead of SAST tools) are less effective than CODEQL at finding code defects.

Finding 10: Recent work has shown that developers will tolerate a reasonable level of false positives, even far exceeding the traditional bound of ~20% [20]. The false positive rate of CODEQL meets developers’ requirements — we sampled 468 of its warnings and measured a false positive rate of only 23%.

8 LIMITATIONS AND THREATS TO VALIDITY

We acknowledge limitations and threats to the validity of our study.

Construct Validity: We scope the construct of security vulnerabilities to those detectable by the SAST tools from the GitHub Marketplace. Other classes of security vulnerabilities exist but are beyond the scope of our work.

Internal Validity: This work was a measurement study, and we made no causal inferences.

External Validity: Our methodology applies the 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-grade ones such as Coverity and Sonar. Our results may not generalize to other embedded software, particularly commercial-grade embedded software to which costly techniques such as formal methods have been applied. To shed light on this threat, in our analysis, we reported on the subset of commercially-developed open-source software, such as Amazon's `aws/aws-iot-device-sdk-embedded-c` and, and show that SAST tool was able to find defects. 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 (Section 5.3) may not be generalizable to other EMBOSS repositories. As a mitigation, the response rate was consistent with other surveys of GitHub developers.

9 DISCUSSIONS AND FUTURE WORK

We were surprised to find many security defects across various embedded software by using an existing SAST tool. In retrospect, these results could be anticipated as most EMBOSS repositories do not use SAST tools. Developers expressed concerns over the benefit and false positive rates of SAST tools. However, most of the defects found by SAST tools are acknowledged and fixed by developers; this shows that SAST tools can find important defects. Compilation required SAST tools, such as CODEQL, cannot handle the diverse build setups of EMBOSS repositories and consequently might fail to run. However, these tools can be easily configured to run with minimal engineering effort. We created GitHub Workflows, which can serve as templates that developers can use to run compilation required SAST tools effectively.

In summary, our results provide a solid case for the need to use standard SAST tools in EMBOSS repositories. Our results also complement a recent work [22] that used simple systematic testing to find several severe security issues in popular embedded network stacks. The research and engineering communities need to enable and adopt well-known techniques on embedded software. As part of our future work, we will work on improving CODEQL to improve its plug-and-play performance and its query precision on embedded repositories.

10 RELATED WORK

In Section 2 and Section 3 we discussed directly related work. Here we compare to more broadly related work.

Embedded Operating Systems and Frameworks: Al-Boghdady *et al.* [12] conducted a thorough analysis of four IoT Operating Systems, namely RIOT [9], Contiki [7], FreeRTOS [19], and Amazon FreeRTOS [18]. Their results indicated an increasing trend in the number of security errors over time. However, the error density remained stable or showed a minor decrease. Alnaeli *et al.* [15, 16] focused their investigation on Contiki and TinyOS, finding an increase in the use of unsafe statements over five years. Meanwhile, McBride *et al.* [84] analyzed the Contiki operating system and found that while errors increased over time, error density decreased. Malik *et al.* [77] carried out a study on embedded frameworks, evaluating security vulnerabilities from four popular edge frameworks.

Their findings revealed that vulnerabilities often slipped through during development due to the challenges of in-house testing of complex Edge features.

Other Analyses of Embedded Systems: Bagheri *et al.* [28] proposed a method for automatically generating assurance cases for software certification. Jia *et al.* [64] propose ContextIoT, a context-based permission system that instruments IoT apps to log fine-grained control and data flow context in order to distinguish malicious behaviors in a robust manner. Celik *et al.* [34] present SOTERIA, a system that applies static analysis and model checking to automatically analyze IoT apps and environments for security and safety violations. They extract a state model from IoT source code and use a model checker to validate desired properties. Evaluation on real-world SmartThings [96] apps shows SOTERIA can effectively identify security and functionality issues in both individual apps and multi-app environments. Our work broadens the scope of these studies. We analyze not only several operating systems but also a diverse corpus of embedded software, highlighting the challenges and effectiveness and offering a more comprehensive and holistic view of the security landscape in IoT systems.

Developers' Perspectives on SAST Tools: Johnson *et al.* [65] found that while developers are aware of the benefits, false positives and the presentation of warnings act as barriers. Our study revealed slightly different findings. In addition to false positives, developers were unaware of the effectiveness of SAST tools on embedded software. However, Johnson *et al.* [65] did not provide any insight into the adoption rates of static analysis tools. In a similar vein, Ami *et al.* [20] conducted in-depth semi-structured interviews with 20 practitioners to shed light on developer perceptions and desires related to static analysis tools. They considered these tools to be highly beneficial in reducing developer effort and covering areas that manual analysis might overlook. 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. Our experiments with CODEQL (an effective SAST tool) showed contradictory results. 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.

11 CONCLUSIONS

We conducted an empirical investigation on the use of static analysis tools in open-source embedded software. We found that there are many free plug-and-play static analysis tools that can be integrated into analysis workflows, but that most of them do not perform well on embedded software. *However, CODEQL is effective.* With minimal engineering cost, we configured and ran it on 258 popular embedded software projects on GitHub. We found 540 defects (with a false positive rate of only 23%), 273 of which have been confirmed. This includes 219 defects that are security vulnerabilities such as crashes and memory corruption. The primary difficulty we observed in the process was configuring diverse build systems, but this took minimal engineering effort per project. 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 continue to push the bounds of vulnerability discovery, but effort must be made to promote adoption.*

REFERENCES

- [1] [n. d.]. Application Security Testing. <https://www.gsa.gov/technology/it-contract-vehicles-and-purchasing-programs/technology-products-services/it-security/application-security-testing>
- [2] [n. d.]. OSRTOS. <https://www.osrtos.com/>.
- [3] [n. d.]. Pull Request for Improvements to CodeQL. REDACTED.
- [4] 2021. mogwailabs finds bugs using CodeQL. <https://mogwailabs.de/en/blog/2021/09/vulnerability-digging-with-codeql/>.
- [5] 2022. Trail of Bits finds bugs using CodeQL. <https://blog.trailofbits.com/2022/01/11/finding-unhandled-errors-using-codeql/>.
- [6] 2023. Computer Security: Avoiding salmonella in your code. <https://rb.gy/yky2e>
- [7] 2023. The Contiki Operating System. <https://github.com/contiki-os/contiki>. original-date: 2012-10-24T05:59:36Z.
- [8] 2023. Node.js. <https://github.com/nodejs/node>. original-date: 2014-11-26T19:57:11Z.
- [9] 2023. RIOT - The friendly Operating System for the Internet of Things. <https://www.riot-os.org/>
- [10] 2023. SwiftSyntax. <https://github.com/apple/swift-syntax>. original-date: 2018-07-31T23:19:58Z.
- [11] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. 2019. Challenges in designing exploit mitigations for deeply embedded systems. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 31–46.
- [12] Abdullah Al-Boghday, 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). <https://doi.org/10.3390/s21072329>
- [13] 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. <https://doi.org/10.1109/COMST.2020.2988293> Conference Name: IEEE Communications Surveys & Tutorials.
- [14] 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.
- [15] 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. <https://doi.org/10.1109/WF-IoT.2016.7845497>
- [16] 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. <https://doi.org/10.25046/aj0203188>
- [17] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. *Proceedings - IEEE Symposium on Security and Privacy* 2019-May (2019), 1362–1380.
- [18] Amazon Web Services, Inc. or its affiliates. 2023. *Amazon FreeRTOS*. <https://aws.amazon.com/freertos/>
- [19] Amazon Web Services, Inc. or its affiliates. 2023. *FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions*. <https://www.freertos.org/index.html>
- [20] 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 *Proceedings of the 2024 IEEE Symposium on Security and Privacy (S&P)*. To appear.
- [21] 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.
- [22] Paschal Amusuo, Andres Calvo Mendez Ricardo, Zhongwei Xu, Aravind Machiry, and James Davis. 2023. Systematically Detecting Packet Validation Vulnerabilities in Embedded Network Stacks. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*.
- [23] Paschal C. Amusuo, Aishwarya Sharma, Siddharth R. Rao, Abbey Vincent, and James C. Davis. 2022. Reflections on Software Failure Analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1615–1620. <https://doi.org/10.1145/3540250.3560879>
- [24] 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.
- [25] 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.
- [26] ANONYMOUS AUTHOR(S). 2023. An Empirical Study on the Use of Static Analysis Tools in Open Source Embedded Software (Supplementary Material). Google Drive. https://drive.google.com/file/d/1kH5UEw2LmXQkKbFLa7mb5k4grHtYU0_/view?usp=sharing
- [27] 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)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

- [28] 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)* (2020-10), 61–64.
- [29] Pranshu Bajpai and Adam Lewis. 2022. Secure Development Workflows in CI/CD Pipelines. In *2022 IEEE Secure Development Conference (SecDev)*. IEEE, 65–66.
- [30] 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.
- [31] Paul E Black and Paul E Black. 2018. *Juliet 1.3 test suite: Changes from 1.2*. US Department of Commerce, National Institute of Standards and Technology.
- [32] Guillaume Brat, Jorge A Navas, Nija Shi, and Arnaud Venet. 2014. IKOS: A framework for static analysis based on abstract interpretation. In *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings 12*. Springer, 271–277.
- [33] Carnegie Mellon University. 2020. OOP50-CPP. Do Not Invoke Virtual Functions from Constructors or Destructors - SEI CERT C++ Coding Standard - Confluence. <https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP50-CPP.+Do+not+invoke+virtual+functions+from+constructors+or+destructors>.
- [34] 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. <https://www.usenix.org/conference/atc18/presentation/celik>
- [35] George Chatzileftheriou 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*. IEEE, 96–103.
- [36] Checkmarx Ltd. 2023. *Checkmarx*. <https://checkmarx.com/>
- [37] Ben Chelf and Christof Ebert. 2009. Ensuring the Integrity of Embedded Software with Static Code Analysis. *IEEE Software* 26, 3 (May 2009), 96–99. <https://doi.org/10.1109/MS.2009.65>
- [38] cpp-linter. 2023. C/C++ Linter Action | Clang-Format & Clang-Tidy. *cpp-linter*.
- [39] 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)*. 338–348. <https://doi.org/10.1109/SANER53432.2022.00050>
- [40] 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*). Association for Computing Machinery, New York, NY, USA, 156–159. <https://doi.org/10.1145/3543873.3587336>
- [41] 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.
- [42] Jacob Domagala. 2023. Static Analysis.
- [43] DOMARS. 2023. CodeQL and the Static Tools Logo Test - Windows drivers. <https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/static-tools-and-codeql>.
- [44] 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*. 687–701.
- [45] 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*). Association for Computing Machinery, New York, NY, USA, 113–122. <https://doi.org/10.1145/2737182.2737190>
- [46] The Linux Foundation. 2023. Open Source Security Foundation (OpenSSF). <https://openssf.org/>.
- [47] Frama-C. 2023. Frama-C/Eva: Sound Analysis of Possible Runtime Errors. Frama-C.
- [48] 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. <https://doi.org/10.1109/TSP.2018.8441496>
- [49] Christoph Gentsch. 2020. Evaluation of open source static analysis security testing (SAST) tools for c. (2020).
- [50] GitHub, Inc. 2021. CodeQL Wall of Fame. <https://securitylab.github.com/codeql-wall-of-fame/>.
- [51] GitHub, Inc. 2023. About code scanning. <https://rb.gy/62fqg>.
- [52] GitHub, Inc. 2023. CodeQL Action. GitHub.
- [53] GitHub, Inc. 2023. CodeQL documentation. <https://codeql.github.com/docs/>.
- [54] GitHub, Inc. 2023. CodeQL Query Help for C and C++ — CodeQL Query Help Documentation. <https://codeql.github.com/codeql-query-help/cpp/>.
- [55] GitHub, Inc. 2023. GitHub Acceptable Use Policies - GitHub Docs. <https://rb.gy/vlqaw>.
- [56] GitHub, Inc. 2023. GitHub Marketplace · Actions to improve your workflow. <https://github.com/marketplace?category=&query=&type=actions&verification=>.

- [57] GitHub, Inc. 2023. Usage limits, billing, and administration. <https://docs.github.com/en/actions/learn-github-actions/usage-limits-billing-and-administration>.
- [58] 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.
- [59] Nikhil Krishna Gopalakrishna, Dharun Anandayuvraj, 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.
- [60] Jez Humble and David Farley. 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.
- [61] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. 2019. How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverage Usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE) (2019-10)*. 323–333. <https://doi.org/10.1109/ISSRE.2019.00040> ISSN: 2332-6549.
- [62] 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*. 1–3.
- [63] IvanKuchin. 2021. C/C++ SAST.
- [64] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z. Morley Mao, and Atul Prakash. 2017. ContextIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *21st Network and Distributed Security Symposium*.
- [65] 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, 2013-05). IEEE, 672–681. <https://doi.org/10.1109/ICSE.2013.6606613>
- [66] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. 2022. Characterizing the Security of Github {CI} Workflows. In *31st USENIX Security Symposium (USENIX Security 22)*. 2747–2763.
- [67] Konstantin343. 2022. CppCheck Annotation Action.
- [68] Brenno Lemos. 2023. C/C++ Linter Action — Cached | Clang-Format & Clang-Tidy.
- [69] Linearity. 2010. Omitting Return Statement in C++.
- [70] 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*. ACM, Virtual South Korea, 544–555. <https://doi.org/10.1145/3533767.3534380>
- [71] 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. <https://doi.org/10.1109/CHASE.2019.00023>
- [72] 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: 8th International Conference, SATE 2018, Shenzhen, Guangdong, China, November 23–24, 2018, Proceedings 8*. Springer, 51–67.
- [73] Björn Lundell, Brian Lings, and Edvin Lindqvist. 2010. Open Source in Swedish Companies: Where Are We?: Open Source in Swedish Companies. *Information Systems Journal* 20, 6 (2010), 519–535.
- [74] 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), 1540–1549.
- [75] Chujiao Ma, Matthew Bosack, Wendy Rothschild, Noopur Davis, and Vaibhav Garg. [n. d.]. Wanted Hacked or Patched. ([n. d.]).
- [76] 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.
- [77] Jahanzaib Malik and Fabrizio Pastore. 2023. An empirical study of vulnerabilities in edge frameworks to support security testing improvement. 28, 4 (2023), 99. <https://doi.org/10.1007/s10664-023-10330-x>
- [78] 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* 47, 11 (2019), 2312–2331.
- [79] Muskan Mangla. 2023. *Securing CI/CD Pipeline: Automating the detection of misconfigurations and integrating security tools*. Ph. D. Dissertation. Dublin, National College of Ireland.
- [80] Steve Mansfield-Devine. 2018. DevOps: finding room for security. *Network security* 2018, 7 (2018), 15–20.
- [81] Joel Margolis, Tae Tom Oh, Suyash Jadhav, Young Ho Kim, and Jeong Noyo Kim. 2017. An in-depth analysis of the mirai botnet. In *2017 International Conference on Software Security and Assurance (ICSSA)*. IEEE, 6–12.
- [82] Daniel Marjamäki. 2013. Cppcheck: a tool for static c/c++ code analysis. URL: <https://cppcheck.sourceforge.io> (2013).
- [83] Alfi Maulana. 2023. CMake Action - GitHub Marketplace. <https://github.com/marketplace/actions/cmake-action>.

- [84] 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* (Madrid, Spain) (EWSN '18). Junction Publishing, USA, 278–283.
- [85] 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).
- [86] 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 Symposium (NDSS)*.
- [87] 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 (2021), 102470.
- [88] 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. <https://doi.org/10.1016/j.cose.2021.102470>
- [89] 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.
- [90] Open Text. 2023. Fortify. <https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer>
- [91] Dipankar Pal. 2022. Flawfinder-Action.
- [92] Dipankar Pal. 2023. Deep5050/Cppcheck-Action.
- [93] 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)*. IEEE, 340–342.
- [94] J. Pincus and B. Baker. 2004. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security & Privacy* 2, 4 (July 2004), 20–27. <https://doi.org/10.1109/MSP.2004.36>
- [95] 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.
- [96] Samsung Electronics Co., LTD. 2023. SmartThings. <https://www.smarthings.com/>
- [97] Wedy Freddy Santoso and Dadang Syarif Sihabudin Sahid. 2021. Implementation and performance analysis development security operations (DevSecOps) using static analysis and security testing (SAST). *International ABEC* (2021), 17–19.
- [98] 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* (Orlando, FL, USA) (LCTES 2023). Association for Computing Machinery, New York, NY, USA, 143–147. <https://doi.org/10.1145/3589610.3596271>
- [99] 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.
- [100] 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). Association for Computing Machinery, New York, NY, USA, 8–14. <https://doi.org/10.1145/2989238.2989239>
- [101] The Linux Foundation. 2023. Zephyr® Project. <https://www.zephyrproject.org/>.
- [102] Lucas Torri, Guilherme Fachini, Leonardo Steinfeld, Vesmar Camara, Luigi Carro, and Érika Cota. 2010. An Evaluation of Free/Open Source Static Analysis Tools Applied to Embedded Software. In *2010 11th Latin American Test Workshop*. 1–6. <https://doi.org/10.1109/LATW.2010.5550368>
- [103] Linus Torvalds. 2023. torvalds/linux. <https://github.com/torvalds/linux>. original-date: 2011-09-04T22:48:12Z.
- [104] trunk-io. 2023. Trunk.Io GitHub Action. Trunk.io.
- [105] Veracode. 2023. Veracode. <https://www.veracode.com/>
- [106] David Wheeler. 2006. Flawfinder home page. *Web page: http://www.dwheeler.com/flawfinder* (2006).
- [107] whisperity. 2023. CodeChecker C++ Static Analysis Action.
- [108] Elecia White. 2011. *Making Embedded Systems: Design Patterns for Great Software*. "O'Reilly Media, Inc."
- [109] Guest Writer. 2020. The 5 Worst Examples of IoT Hacking and Vulnerabilities in Recorded History. <https://www.ietf.org/5-worst-iot-hacking-vulnerabilities>.
- [110] 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)*. IEEE, 161–164.
- [111] Ruotong Yu, Francesca Del Nin, Yuchen Zhang, Shan Huang, Pallavi Kaliyar, Sarah Zakto, Mauro Conti, Georgios Portokalidis, and Jun Xu. 2022. Building embedded systems like it's 1996. *arXiv preprint arXiv:2203.06834* (2022).