An Empirical Study of Static Analysis Tools for Secure Code Review

Wachiraphan Charoenwet
The University of Melbourne
Australia
wcharoenwet@student.unimelb.edu.au

Patanamon Thongtanunam
The University of Melbourne
Australia
patanamon.t@unimelb.edu.au

Van-Thuan Pham
The University of Melbourne
Australia
thuan.pham@unimelb.edu.au

Christoph Treude
Singapore Management University
Singapore
ctreude@smu.edu.sg

Abstract

Early identification of security issues in software development is vital to minimize their unanticipated impacts. Code review is a widely used manual analysis method that aims to uncover security issues along with other coding issues in software projects. While some studies suggest that automated static application security testing tools (SASTs) could enhance security issue identification, there is limited understanding of SAST's practical effectiveness in supporting secure code review. Moreover, most SAST studies rely on synthetic or fully vulnerable versions of the subject program, which may not accurately represent real-world code changes in the code review process.

To address this gap, we study C/C++ SASTs using a dataset of actual code changes that contributed to exploitable vulnerabilities. Beyond SAST's effectiveness, we quantify potential benefits when changed functions are prioritized by SAST warnings. Our dataset comprises 319 real-world vulnerabilities from 815 vulnerability-contributing commits (VCCs) in 92 C and C++ projects. The result reveals that a single SAST can produce warnings in vulnerable functions of 52% of VCCs. Prioritizing changed functions with SAST warnings can improve accuracy (i.e., 12% of precision and 5.6% of recall) and reduce Initial False Alarm (lines of code in non-vulnerable functions inspected until the first vulnerable function) by 13%. Nevertheless, at least 76% of the warnings in vulnerable functions are irrelevant to the VCCs, and 22% of VCCs remain undetected due to limitations of SAST rules. Our findings highlight the benefits and the remaining gaps of SAST-supported secure code reviews and challenges that should be addressed in future work.

CCS Concepts

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software creation and management**.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680313

Keywords

Code Review, Static Application Security Testing Tool, Code Changes Prioritization

ACM Reference Format:
Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. 2024. An Empirical Study of Static Analysis Tools for Secure Code Review. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680313

1 Introduction

Managing security issues in software products is crucial because such issues, especially exploitable vulnerabilities, can exponentially impact end users and require more resources to resolve if discovered in a later stage. Aligned with the Shift-Left concept, which advocates for the early issue detection in software [71], previous work [33] argues that manual code review is an approach adopted by numerous software projects to identify and mitigate issues before merging new code changes into the existing codebase. Unlike traditional practice, modern code review lacks concrete requirements [31] and can take multiple iterations to complete. The effort that reviewers, who typically have many resource constraints, must invest poses a persistent challenge for secure code reviews (i.e., finding security issues in the reviewed code). Indeed, identifying issues that could contribute to vulnerabilities often requires security knowledge and meticulous inspection from reviewers [33], and hence fortifies the challenges.

Although prior code review studies [51, 56, 61] suggested that static analysis tools could help reviewers identify issues related to coding styles, none have explored their effectiveness as the static application security testing tools (*SASTs*, hereafter). Conversely, current SAST studies [28, 29, 49, 55, 63] have investigated SAST's effectiveness based on the released software versions that contain exploitable vulnerabilities. The existing works have not examined the tools with vulnerabilities that gradually appear during the software development process. In code reviews, changes are typically small [58] containing security issues that are both difficult to identify and easy to overlook [33]. Moreover, a complete vulnerability may require the contribution of code changes from multiple code commits [45]. With the different contexts and nature of code reviews, it remains unclear to what extent SASTs can cost-effectively

assist reviewers in finding security issues. To the best of our knowledge, prior works have not investigated the effectiveness of the SASTs using code commits, and for code reviews.

We investigate the capability of SASTs to support secure code reviews by addressing three key aspects: ① effectiveness of SASTs on vulnerable code commits, ② potential benefits of SAST-supported code reviews, and ③ associated waiting time. Our empirical study uses 815 real-world *vulnerability-contributing commits* (VCCs) in 92 C and C++ software projects. These VCCs involve 1,060 vulnerable functions , contributing to 319 exploitable vulnerabilities. With this dataset, we study five commonly used C and C++ SASTs [49], which employ diverse static analysis techniques to detect vulnerabilities, i.e., Cppcheck, CodeChecker, CodeQL, Flawfinder, and Infer. To measure the effectiveness of SASTs, we analyze how many VCCs the tools can warn about. For the potential benefits of code reviews supported by SASTs, we analyze how many vulnerable functions can be identified within a limited code review effort when changed functions are prioritized based on the information in the SAST warnings [44] (henceforth, *warning-based prioritization)*. To do so, we measure the accuracy using precision and recall at a fixed review effort (i.e., 25% of lines of code in changed functions) and Initial False Alarm (IFA)—percentage of lines of code in warned functions until the first vulnerable functions. Additionally, we measure the time that each tool takes to analyze the VCCs.

Our results show that a single SAST can produce warnings in the vulnerable changes (i.e., changed files and changed functions) of more than half of the VCCs. Specifically, Flawfinder can produce warnings in at least one vulnerable function for 52%, and one vulnerable file for 89%, of 815 VCCs. Combining warnings from multiple SASTs can achieve a higher vulnerability detection rate, i.e., 78% at the function level.

With respect to code reviews, prioritizing changed functions using the warnings from SASTs during code reviews can enhance the discovery of vulnerable changes by increasing up to 12% of precision and 5.6% of recall and reducing up to 13% of IFA at 25% of code review effort. Lastly, the computation time of SASTs should fit within the code review waiting period since the average computation time on the selected VCCs ranges from 20 seconds to 45 minutes, with a notable increase when the size of projects reaches 50k-100k LOC. Nevertheless, SASTs still need an improvement because at least 76% of the warnings are irrelevant to the vulnerability in the corresponding VCCs. Additionally, 22% of VCCs (10% of the exploitable vulnerabilities) do not receive any warnings from any of the five studied tools.

Novelty and Contribution: To the best of our knowledge, this paper is the first to:

- Investigate the effectiveness of C and C++ SASTs on the code commits, which are subject to code reviews, that introduced real-world vulnerabilities in 92 projects of diverse application domains, covering eight types of CWE vulnerabilities
- Investigate the potential benefits of warning-based prioritization for vulnerability identification in VCCs' changed functions during code reviews
- Provide a versatile and extensible automated benchmarking framework for SAST evaluation using open-source code commit datasets in various programming languages [35]
- Release an annotated dataset of 1,064 vulnerable files and 1,060 vulnerable functions in 815 commits [36]

Paper Organization: The rest of this paper is organized as follows: Section 2 explains the background of this study. Section 3 presents our study design. Section 4 presents our study approaches and results of each research question. Section 5 discusses the findings. Section 6 reports threats to validity. Section 7 discusses related work. Finally, Section 8 draws the conclusion.

2 Background and Definitions

In this section, we briefly describe the key concepts and terms used in this paper and discuss the limitations and challenges.

Secure code review refers to a code review practice that focuses on security aspects of the software system. This practice conforms to the shift-left concept that promotes early quality assurance in software development [71] to reduce the impacts of undiscovered security issues. During the development cycle, developers submit a code commit that contains small code changes [58] to be reviewed by other developers before it can be integrated into the codebase. Reviewers can inspect the code changes and provide feedback on various aspects, including security [64], functionality, and maintainability [31, 50]. The authors can modify the submission to address the reviewers' feedback. These activities continue until the code changes are accepted and merged into the codebase. Nevertheless, despite the widespread code review practices, identifying security issues can be challenging for reviewers. Recent studies on secure code review [33, 34] highlighted that manually identifying security issues in the code is difficult because reviewers may not have the required knowledge and sufficient awareness of possible issues.

Vulnerability contributing commit (VCC) is a set of code changes that encompass the modified code contributed to a vulnerability. It represents the subject of the code review process, which aims to uncover the vulnerabilities during the development cycle. Iannone et al. [45] conducted a large-scale study on the vulnerability lifecycle, revealing that real-world vulnerability can be composed of multiple commits over time. For instance, an *SQL injection* requires an average of 18 VCCs. It can be implied that a vulnerability can potentially be prevented if the issues in VCC are identified by the reviewers, and fixed by the developers, during code review.

Static application security testing tool (SAST) is a quality assurance tool that identifies certain security-related issues in source code using the pre-defined set of rules without executing the program. We consider SASTs that employ two overarching analysis techniques [49]: ① syntactic-based and ② semantic-based. While many studies [29, 39–41, 48, 49, 57, 60] empirically evaluated SASTs on identifying security issues, they neglect the context of code reviews and VCCs. For example, Lipp et al. [49] evaluated the effectiveness of tools with a partially synthetic dataset, and a recent study by Li et al. [48] evaluated the tools on the vulnerable versions of real-world programs. The key differences between our work and recent SAST studies are shown in Table 1. While previous studies related to code reviews [51, 56, 61] investigated the capabilities of SASTs in helping reviewers identify coding styles or patterns, there are still knowledge gaps about the capabilities of SASTs for

Table 1: Key differences between this work and the recent SAST studies focusing on vulnerability identification.

Study	Lang.	Subject Projects	CVEs	CWE Groups	Exec. Time	Code Commits
Li et al. [48]	Java	63	165	7	✓	-
Lipp et al. [49]	C/C++	9	192	5	✗	-
Our Study	C/C++	92	319	8	✓	815 VCCs

vulnerability identification. Specifically, the following aspects have not been incorporated into the existing studies: ① the context of code changes, e.g., modified files and functions, ② the benefits of code reviews supported by SAST warnings, and ③ the amount of time that reviewers need to wait for SAST results.

3 Study Design

To bridge the knowledge gaps of the SASTs-supported secure code reviews, we study SASTs using the *real-world vulnerability contributing commits*. Evaluating SASTs with this dataset will help researchers and practitioners better understand the capabilities of the tools to assist reviewers in identifying potential vulnerabilities during the code review process. We also investigate SASTs' computing time.

In the following, we present our research questions and the study design. The overview of the study process is shown in Figure 1.

3.1 Research Questions

In this study, we set out to answer the following research questions.

RQ1 Effectiveness: How effectively can SASTs detect vulnerabilities in vulnerability contributing commits?
Motivation: Previous work [29, 39–41, 57, 60] studied SASTs with synthetic benchmark datasets or certain versions of programs that contain complete vulnerabilities. The existing datasets may not represent the vulnerabilities that gradually appear from code changes during the development process. Investigating SASTs using VCCs should extend the understanding of the tool's capability to detect vulnerabilities in this context.
Approach: We execute SASTs on the selected VCCs and determine the number of VCCs that SASTs can produce warnings on vulnerable code changes. In this work, we consider the scope of code changes in VCCs at the file and function levels. We also analyze the types of vulnerabilities in VCCs that SASTs can detect.

RQ2 Warning-based Prioritization: How can we use SAST warnings to prioritize changes for secure code reviews?
Motivation: Reviewers have limited review capacity [33, 61]. Previous work [30, 32, 43] argued that SAST warnings may help reduce reviewer effort. Muske and Serebrenik [53] reported that developers prioritize the SAST warnings that they should pay attention to. Thus, we hypothesize that warning-based prioritization of changed functions could reduce reviewing effort.
Approach: We set out to investigate whether, within a limited review effort, warning-based prioritization can help reviewers identify more vulnerable functions. We measure the accuracy of identifying vulnerable functions prioritized by SAST warnings within a fixed review effort (i.e., $K\%$ lines of code in changed functions).

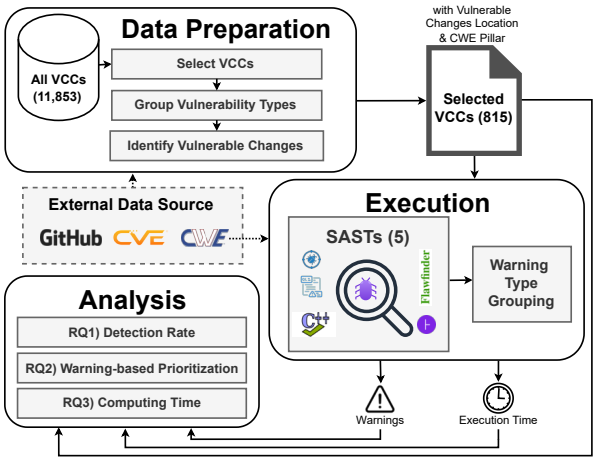


Figure 1: Overview of our study approach.

RQ3 Waiting Time: How much computation time is required?
Motivation: Kudrjavets et al. [47] suggested that automated tests should be executed during the *non-productive* period of code reviews. To incorporate SASTs without delaying code reviews, the computation time should fit this idle period. However, the computation time of C/C++ SASTs on VCCs remains underexplored.
Approach: We measure the computation time from when the tool starts analyzing a commit until the warnings are produced. Additionally, we analyze the computation time by the system's lines of code to understand whether the time depends on system size [48].

3.2 Data Preparation

Since the benchmark datasets from prior studies [40, 49] are obtained from released versions, they may not represent the nature of code changes in the software development and code review processes where the changes are small and may be mixed with other changes. Thus, a new dataset of code changes contributing to exploitable vulnerabilities is more suitable. We describe the data preparation process for our study below.

3.2.1 Dataset. To craft the dataset, we begin with the Vulnerability Contributing Commits (VCCs) dataset provided by Iannone et al. [45]. This dataset is suitable for our study because it collected real-world code commits that have been identified as contributing to exploitable vulnerabilities. The VCC datasets were collected from GitHub projects associated with vulnerabilities in the Common Vulnerabilities and Exposures (CVE) database [3]—the collection of exploitable vulnerabilities in software systems that are publicly reported by the community. Given a fixing commit recorded in a CVE, a VCC was identified using the SZZ algorithm [62]. Each VCC includes meta-information, i.e., the repository URL, commit identification number (Commit SHA), related vulnerability (CVE number), and related vulnerability type (CWE item). It should be noted that the VCC dataset does not provide the actual changes in each commit, e.g., vulnerable files or vulnerable functions.

3.2.2 Selecting VCCs. Although the VCC dataset of Iannone et al. [45] is sizeable, some VCCs cannot be analyzed by SAST because of

Table 2: Vulnerability types of selected VCCs. A single VCC can be related to more than one CWE pillar.

Short Name	CWE Pillar	#VCCs
Access Ctrl	**CWE-284** Improper Access Control†	38
Resource Ctrl	**CWE-664** Improper Control of a Resource Through its Lifetime	538
Incorrect Cal	**CWE-682** Incorrect Calculation	78
Control Flow	**CWE-691** Insufficient Control Flow Management	56
Protect Mech	**CWE-693** Protection Mechanism Failure†	13
Cond Check	**CWE-703** Improper Check or Handling of Exceptional Conditions	20
Neutalization	**CWE-707** Improper Neutralization	56
Coding Std	**CWE-710** Improper Adherence to Coding Standards†	45

† indicates the additional pillars from [49]

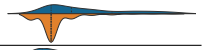
incompatible programming languages or compilation issues; others may have insufficient information for our analysis (e.g., missing vulnerability type). We select VCCs based on the following criteria:

(1) **From C/C++ projects:** We selected VCCs from the projects implemented in C or C++ because these languages allow the programs to interact with low-level operations that can lead to exploitable vulnerabilities [67]. To determine this, we obtained the project's main programming language from GitHub API.
(2) **Have an assigned vulnerability type:** Since we will analyze the vulnerability type that SASTs can detect, we must ensure that the selected VCCs have sufficient vulnerability information. The associated CVE of a VCC is usually assigned a vulnerability type in the Common Weakness Enumeration (CWE) taxonomy. Thus, we selected VCCs with at least one CWE item assigned to the associated CVE.
(3) **Compilable:** Some SASTs require the information from the compilation process. Thus, the selected VCCs must be compilable. However, diverse build processes and dependencies across many C/C++ projects in the VCC dataset pose challenges for automated compilation tests, increasing the chance of build failure. Manually compiling every VCC is also infeasible due to the large number of VCCs. To mitigate the issue, we first selected the projects that use the standard build process, i.e., `GNU Autotools` as it has been used by many C and C++ projects (over 38%) in the original VCC dataset. Then, we selected VCCs that can be compiled successfully following the instructions in the project's document.

Based on the first criterion, we obtained 5,354 VCCs in 341 projects from the VCC dataset. Then, using the second criterion for the obtained VCCs, we identified 5,015 VCCs linked to 1,654 CVEs with at least one CWE item. Finally, 1,057 VCCs with 371 CVEs from 92 projects passed the compilation test in our third criterion.

3.2.3 Vulnerability Type Grouping. We used vulnerability information in a *CWE item* to analyze the vulnerability type in this study. CWE items represent certain weaknesses that can lead to vulnerabilities in a software system. However, while the CWE items enable detailed analyses of individual vulnerabilities, they do not embody the similar vulnerabilities in the VCC dataset that are assigned with different CWE items. Therefore, we grouped the assigned CWE items of VCCs to a *CWE pillar*, which is the top level of the CWE hierarchy, representing abstract categories of the CWE items [40]. Note that we do not use other CWE grouping methods such as

Table 3: Demographics of code changes in selected VCCs. The mean and median values compare the central tendencies between each pair of items.

Description	Mean	Median	Distribution
CVE Vulnerabilities (319)			
VCCs Contributed to CVE	2.5	2	
VCCs with Vulnerable Files (815)			
Changed Files (9,851)	12	3	
Vulnerable Files (1,064)	1	1	
LOC in Changed Files	9,956	2,492	
LOC in Vulnerable Files	1,820	1,034	
VCCs with Vulnerable Functions (697)			
Changed Functions (34,541)	38.9	6	
Vulnerable Functions (1,060)	1.5	1	
LOC in Changed Functions	3,304	531	
LOC in Vulnerable Functions	221	114	

CWE categories [15] because they only cover 400 CWE items out of over 900 CWE items in the ten pillars of CWE hierarchy. To identify the associated pillar of a CWE item, we traced the CWE item's parent to the root of the tree. Figure 2 illustrates our grouping approach. For example, CWE items `Use of Uninitialized Variable` (CWE-457) [16] and `Expired Pointer Dereference` (CWE-825) [17] will be grouped into the `Improper Control of a Resource Through its Lifetime` (CWE-664) pillar [18].

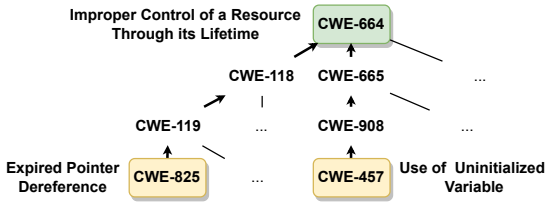


Figure 2: An illustrative example of grouping CWE items (CWE-825 and CWE-457) to the CWE pillar (CWE-664).

In total, our dataset covers eight CWE pillars, which is more diverse than the benchmark dataset of prior work [49] (Table 1). Table 2 shows the number of VCCs in each of the CWE pillars.

3.2.4 Identifying Vulnerable Changes Location in VCCs. Since we aim to investigate whether SASTs can provide warnings to the right locations of vulnerable code, we must identify the scope of vulnerable code in a VCC. We considered two granularity levels, i.e., changed files and changed functions, in VCCs. We did *not* identify the vulnerable changes at the line level because modified lines in fixing commits may not directly correspond to vulnerable lines in VCCs [59]; hence, the reliability can be reduced. However, we also manually evaluated SAST warnings to understand their actual relevancy to the VCCs in subsequent analysis (see Section 5.1).

Using similar intuition as in previous work [62], we considered that a changed file (or a changed function) is vulnerable if it was modified in the associated vulnerability-fixing commits. Following the approach of [45], we obtained the fixing commits from GitHub hyperlinks in the CVE records of VCCs. The changed files in VCCs are considered vulnerable (***vulnerable files***, henceforth) when these files were also changed in the fixing commits. For

the changed functions, we used `lizard` package [27] to identify changed functions in a VCC and the associated fixing commits. The changed functions in VCCs are considered vulnerable (***vulnerable functions***, henceforth) when these functions were also changed in the fixing commits.

We successfully located vulnerable code changes in 815 VCCs where at least one source code file is modified in both VCCs and fixing commits. These VCCs contributed to 319 exploitable vulnerabilities in the CVE database. Table 3 shows the demographics of selected VCCs, emphasizing that the vulnerable changes can be difficult to identify because they are typically small compared to all code changes in each VCC. We use these VCCs in our analysis.

3.3 Execution

This section describes the studied SASTs, experiment setup, and warning type grouping.

3.3.1 Studied Tools. We aim to analyze SASTs that are generally available for C and C++ OSS projects. Specifically, we select SASTs that are ① available for use free of charge, ② able to run on the command-line interface (CLI) as we need to automate the compilation process, and ③ actively maintained by the developers (been updated in the last 12 months).

To select SASTs, we first obtain 53 SASTs from prior studies [49, 54, 63] and NIST's Software Assurance Metrics And Tool Evaluation (SAMATE) [21] as candidates.[1] Then, we examine the documentation of each tool to understand its specifications. From the initial list, we remove 12 SASTs that do not support C and C++ languages. Then, we exclude 26 commercial SASTs, two SASTs that do not operate on the command-line interface, and four SASTs that are not actively maintained. We also omit four SASTs, including those that operate on an external server, are compiler extensions (not employing further rules) or lack information on available warnings. Finally, three semantic-based tools, i.e., CodeChecker[23], CodeQL [24], and Infer [20]; one syntactic-based tool, i.e., Flawfinder [19]; and one hybrid tool, i.e., Cppcheck [25] pass our criteria.

3.3.2 Experiment Setup. In this study, we use the latest stable versions of the selected SASTs, i.e., Cppcheck v2.10, CodeChecker v6.22.2, CodeQL v2.13.3, Flawfinder v2.0.19, and Infer v1.10. For each tool, we use the settings recommended by its developers for reliable results (as also suggested by [52]). In particular, for Flawfinder and Cppcheck, we use the readily enabled rules. For CodeQL, we use the LGTM query suite [13], which contains the largest number of rules. For CodeChecker and Infer, we use the recommended settings. Note that we do not enable all rules for CodeChecker and Infer as it is discouraged by the tools' developers.[2]

Pipeline - We develop an automated framework to facilitate SAST execution. Each VCC is downloaded and prepared before running SAST. The preparation steps are different between SASTs. For SASTs that require compilation (i.e., CodeChecker, CodeQL, and Infer), we update `Makefile` by running the automated script that the projects have prepared for the compilation, i.e., `autogen.sh`, `bootstrap.sh`, and `build.sh`. If none of the scripts is present, we use the `autoreconf` command to generate the required files. Then, we run the configuration file, i.e., `configure`. After the required preparation, we execute SAST and collect the produced warnings for further analysis. Note that the time spent on preparation is also included in the computation time.

The pipeline operates in isolated Docker containers to control the environment and resources for each experiment (i.e., automatically running SAST on a selected set of target code commits). It stores the produced warnings on the host machine for ease of access and manages a centralized database that collects execution status, number of warnings, and timestamps for further analysis.

Infrastructure - To run our experiments, we use a virtual machine with a 32-core virtual CPU and 288 GB of memory. We run each of the studied SASTs in an Ubuntu 22.04 Docker container with a 4-core virtual CPU. Due to the time constraint, we terminate an execution when it continues for more than five hours on any VCCs.

3.3.3 Warning Type Grouping. Since we aim to examine whether a SAST provides warnings that match the vulnerability type of a VCC, we mapped and grouped warnings of the studied SASTs into CWE pillars. For Cppcheck, CodeQL, and Flawfinder, each of their warnings has a CWE item assigned. Thus, we used the same approach described in the vulnerability grouping to group the CWE items into CWE pillars (see Section 3.2.3).

Two SASTs, i.e., CodeChecker and Infer, do not assign CWE items in the warnings. Thus, the warnings produced by these tools must be mapped to the most relevant CWE pillars. We used the mapping between warnings and CWE items provided by Lipp et al. [49]. However, the mapping [49] does not cover all warning types in our VCCs, which contain more diverse vulnerabilities. We consult the official documents [14, 22, 26] to map the remaining warnings without CWE items. For example, we assign CWE item *Reachable Assertion* (CWE-617) to the `bugprone-assert-side-effect`[3] warning of CodeChecker. Out of 178 warnings that needed the new mapping, 5% cannot be linked to any CWE items.[4] The mapping process is conducted primarily by the first author. To ensure the accuracy of the mapping process, the third author, who has over 10 years of experience in software security testing, joined the first author to establish the foundational framework and assisted with ambiguous warnings encountered along the process. Finally, the CWE items are grouped into the CWE pillar as described in Section 3.2.3.

4 Analyses & Results

We report the results of our analysis and answer the research questions in this section.

4.1 Detection Effectiveness (RQ1)

Approach: To answer our RQ1: *How effectively can SASTs detect vulnerabilities in vulnerability contributing commits?*, we measure how many VCCs that SASTs can produce the warnings on the vulnerable changes as well as analyze the types of vulnerabilities. Specifically, we execute the studied SASTs on the 815 selected

[1] The exhaustive list of assessed SASTs is included in the data package

[2] https://codechecker.readthedocs.io/en/latest/analyzer/user_guide/#enable-all and https://github.com/facebook/infer/issues/1114#issuecomment-506284374

[3] https://clang.llvm.org/extra/clang-tidy/checks/bugprone/assert-side-effect.html

[4] We include the complete mappings in our data package

Table 4: Name and description of detection scenarios in two change levels.

Scenario Name by Change Level		Description
File	Function	
S1:*1F-Any*	S5:*1Fn-Any*	At least one vulnerable *change* in the VCC receives warnings with any type.
S2:*1F-Same*	S6:*1Fn-Same*	At least one vulnerable *change* in the VCC receives warnings with the same vulnerability type as the VCC.
S3:*allF-Any*	S7:*allFn-Any*	All vulnerable *changes* in the VCC receive warnings with any type.
S4:*allF-Same*	S8:*allFn-Same*	All vulnerable *changes* in the VCC receive warnings with the same vulnerability type as the VCC.

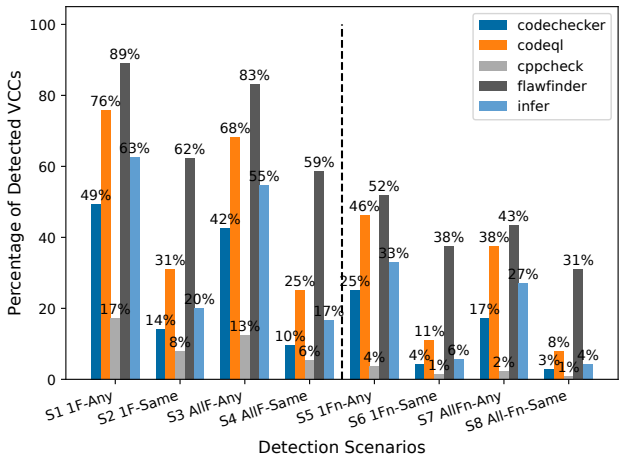


Figure 3: Percentages of VCCs that the tools can detect in different scenarios.

VCCs and examine whether the tools produce warnings to the vulnerable file and vulnerable functions in the VCCs. Similar to prior works [48, 49], for each granularity (i.e., file or function level), we consider the detection rate based on four scenarios (see Table 4).

Results: We consider the results in three aspects: 1) detection rate, 2) detected vulnerabilities by type, and 3) undetected vulnerabilities. **Detection rate:** Figure 3 shows the percentages of VCCs that an SAST can detect in each scenario. Flawfinder produced warnings in vulnerable files and functions of the largest number of VCCs. Hence, it potentially offers the lower false negatives i.e., a low number of vulnerable files and functions that do not receive a warning. Specifically, Flawfinder produces warnings in at least one vulnerable file (S1:*1F-Any*) for 89% of the VCCs. At the function level, which has a smaller scope of code changes, Flawfinder produces warnings in at least one vulnerable function (S5:*1Fn-Any*) in 52% of VCCs. In contrast, Cppcheck produced warnings in vulnerable files and functions of the smallest number of VCCs in all eight scenarios.

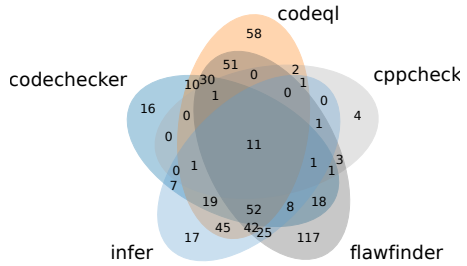


Figure 4: A Venn diagram displaying VCCs for which the tools can produce warning(s) in the vulnerable functions (S5:*1Fn-Any*)

> **Finding (detection rate)**: Flawfinder can produce warnings in the vulnerable changes in 89% of the VCCs at the file level and in 52% of the VCCs at the function level.

We further investigate whether combining SASTs can improve the detection rate. It is possible that different SASTs can detect different vulnerability types. We examine the detection rate of multiple SASTs at the function level *(S5: 1Fn-Any)* as Lipp et al. [49] remarked that function-level is an appropriate granularity for analyzing SAST warnings. Figure 4 shows that multiple tools (i.e., CodeChecker, CodeQL, Flawfinder, and Infer) can commonly detect 52 VCCs, which account for only 6%. This suggests that the tools do not regularly detect the same VCCs. In that respect, when all warnings of the five tools are considered, they collectively detect 541 VCCs (78%), which is substantially increased by 26 percentage points compared to the detection rate in S5:*1Fn-Any* of a single tool with the highest detection rate (i.e., Flawfinder). This highlights that combining these tools enhances their effectiveness in identifying a larger number of VCCs.

> **Finding (combining tools)**: Combining multiple SASTs can increase the effectiveness in identifying vulnerability in VCCs by 26%.

Detected vulnerabilities by type: SASTs can produce warnings on VCCs of almost all vulnerability types in our dataset. Table 5 shows the numbers and percentages of VCCs that SASTs produce warnings with the matching type of vulnerability in at least one vulnerable function (S6:*1Fn-Same*) and all vulnerable functions (*S8: allFn-Same*) by CWE pillars.

- **Flawfinder** warns most VCCs related to **Resource Ctrl** (CWE-664) (38%), e.g., possible buffer overflow [10] and **Neutralization** (CWE-707) (16%), e.g., potential command injection from the insufficient input validation [5].
- **CodeQL** warns most VCCs related to **Access Ctrl** (CWE-284) (3%), e.g., potential information leak [12] ; **Incorrect Cal** (CWE-682) (12%), e.g., the potential DoS caused by incorrect calculations [4] ; and **Control Flow** (CWE-691) (4%), e.g., infinite loop caused by the incorrect data type of loop variable [6].

Table 5: Numbers of VCCs that receive warnings in all vulnerable functions (S8:*allFn-Same*) or at least one vulnerable function (S6:*1Fn-Same*) and percentages by CWE pillars. The first and the second numbers are from S8 and S6, respectively.

CWE Pillar	CodeChecker	CodeQL	Cppcheck	Flawfinder	Infer
Access Ctrl CWE-284	-	**1/1 (3%/3%)**	-	-	-
Resource Ctrl CWE-664	17/44 (3%/4%)	43/63 (8%/12%)	4/8 (1%/1%)	**205/249 (38%/46%)**	24/33 (4%/6%)
Incorrect Cal CWE-682	-	**9/11 (12%/14%)**	1/1 (1%/1%)	2/2 (2%/2%)	-
Control Flow CWE-691	-	**2/2 (4%/4%)**	-	-	-
Cond Check CWE-703	**4/4 (20%/20%)**	-	-	-	-
Neutralization CWE-707	3/3 (5%/5%)	-	-	**9/12 (16%/21%)**	-
Coding Std CWE-710	**8/12 (16%/27%)**	2/2 (4%/4%)	1/1 (2%/2%)	-	6/7 (13%/16%)

- **CodeChecker** warns most VCCs related to **Cond Check** (CWE-703) (20%), e.g., null pointer dereferences caused by the incorrect order of arguments [11] and **Coding Std** (CWE-710), (16%) e.g., the improper `if` statement [7].

> **Finding (vulnerability types)**: CodeQL can detect the most types of vulnerability in VCCs (5 out of 8). Only CodeQL can detect vulnerability type **Access Ctrl** (CWE-284) and **Control Flow** (CWE-691), while only CodeChecker can detect vulnerability type **Cond Check** (CWE-703). Flawfinder can detect most VCCs of type **Resource Ctrl** (CWE-664) and **Neutralization** (CWE-707).

Undetected vulnerabilities: 156 VCCs (22%) spread across 33 projects (35%) did not receive any warnings at the vulnerable functions (S5:*1Fn-Any*). Notably, vulnerabilities related to CWE-664, CWE-682, CWE-693, CWE-707, and CWE-710 are the common types (143 out of the 156 VCCs) that were not detected by the studied SASTs, which account for 10% of the studied vulnerabilities. Surprisingly, some of these undetected vulnerabilities are straightforward and should have been identified by SASTs. For instance, the CVE-2018-17294 [9] involves a buffer over-read due to a missing length check in a loop, deviating from the regular coding pattern [2]. Additionally, vulnerabilities related to **Protect Mech** (CWE-693) went unnoticed by SASTs despite the existence of supporting rules. Manual observation revealed that the current rules are strict to particular code patterns of vulnerabilities; a small deviation can make the vulnerability go undetected. As an example, CVE-2017-13083 [8] allows the execution of external code through an insecure connection. Existing CodeQL queries failed to detect these vulnerabilities, as the queries used regular expressions or focused on OpenSSL functions, while the vulnerable code checked HTTP certificates with Windows API.

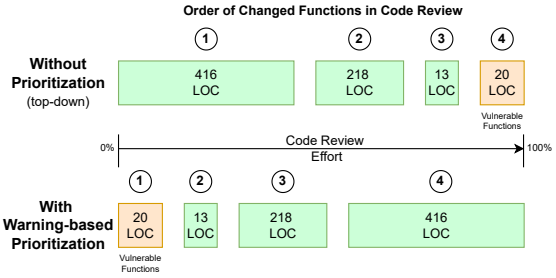


Figure 5: An illustration to depict changed functions without prioritization (above) and with warning-based prioritization (below) using a real VCC from *libsndfile*.

> **Finding (undetected vulnerability)**: 22% of VCCs do not receive warnings in the vulnerable functions. CWE-664, CWE-682, CWE-693, CWE-707, and CWE-710 are the common vulnerability types that the tools miss despite the presence of corresponding rules in the SASTs.

4.2 Warning-based Prioritization (RQ2)

Approach: To answer our RQ2: *How can we use SAST warnings to prioritize changes for secure code reviews?*, we analyze how many vulnerable functions can be identified within a fixed effort (*K*% lines of code; *K*%LOC) when the changed functions in a VCC are prioritized based on warnings of a SAST. Figure 5 illustrates that the warning-based prioritization can affect the order of changed functions in code review of a VCC.[5] Without a prioritization (reviewing changed files alphabetically [1], top-down based on line number in each file), reviewers may stop reviewing the changes before reaching the vulnerable function, or a vulnerable function may be reviewed the last, which requires more review effort to find a vulnerability. In contrast, when a vulnerable function is prioritized by SAST warnings, it may save code review effort in identifying vulnerabilities. We use the warnings from scenario *S5 (1Fn-Any)* to answer this question because warnings may guide reviewers in prioritizing the code changes, although the types differ from the reported vulnerabilities.

We explore three prioritization approaches based on the warning information [66, 68]: ① *Warning Amount (WA)*—a changed function with a high number of warnings reviewed first, ② *Warning Density (WD)*—a changed function with a high number of warnings per LOC reviewed first [66], and ③ *Warning Severity (WS)*—a changed function with high severity warnings[6] reviewed first [68].

Inspired by prior studies [44, 70], we use the following metrics to measure the effectiveness of warning-based prioritization:

- **Initial False Alarm (IFA):** The proportion of lines of code in the changed functions that the reviewer needs to review until

[5] https://github.com/libsndfile/libsndfile/commit/1fc6bb8e0faa130e107f6af98991ac8e4555310b

[6] To facilitate the analysis, we convert the severity levels of warning from each tool to the three levels i.e., High, Medium, and Low. For example, CodeQL's warning severity types *Error*, *Warning*, and *Note* are mapped to *High*, *Medium*, and *Low* respectively. The mapping table is included in the data package.

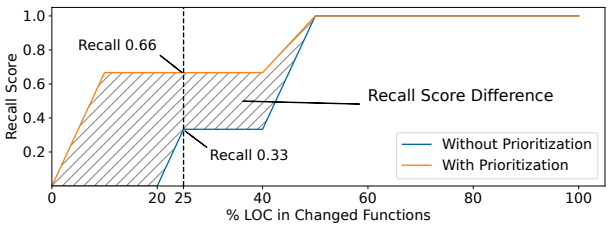


Figure 6: An example of Recall@K%LOC with and without warning-based prioritization of a real VCC from *Leptonica*.

reviewing the first vulnerable function. A high IFA indicates that more effort is needed to review non-vulnerable changes.

- **Precision@K%LOC:** The proportion of vulnerable functions that are prioritized within the top K% LOC compared to the total number of changed functions prioritized within the top$K$%LOC, i.e., $\frac{TP}{TP+FP}$.
- **Recall@K%LOC:** The proportion of vulnerable functions that are prioritized within the top K% LOC compared to the total number of vulnerable functions in a VCC, i.e., $\frac{TP}{TP+FN}$.

We set K=25% LOC for Precision@K%LOC and Recall@K%LOC because the IFA of reviewing changed functions without warning-based prioritization is 25% LOC on average in our VCC dataset, i.e., reviewers typically need to review 25% of LOC of changed functions until the first vulnerable function is reviewed. We also experiment with K=30%, 40%, and 50% LOC to ensure the consistency.

To quantify the improvement of a warning-based prioritization by an SAST, compared to non-prioritization, we measure the performance gain = $\frac{\Sigma_{i \in VCC}(Perf^i_{with} - Perf^i_{without})}{\Sigma_{i \in VCC} Perf^i_{without}}$ [70]. Figure 6 shows an example from a VCC[7] when changed functions are prioritized by the severity levels of CodeChecker's warnings. For example, at K=25%LOC, the warning-based prioritization can improve the recall by 100% ($\frac{0.66-0.33}{0.33}$), compared to a non-prioritization.

In addition, we use the Wilcoxon signed-rank test to confirm whether the performance differences between code reviews with and without tool support across VCCs are statistically different. Since we compare the performance difference between the five tools, we used the Bonferroni correction to calculate the new acceptance threshold at a 95% confidence level, as $\alpha = (\frac{0.05}{5}) = 0.01$.

Results: Table 6 shows that IFA can be reduced by 13% when the functions are prioritized based on CodeQL warnings. When the reviewing effort is fixed at 25% LOC of changed functions, warning-based prioritization can increase precision by 12% when using CodeQL warnings. Recall scores can also be increased by 5.6% when using CodeChecker warnings. Table 6 also shows that the precision and recall of warning-based prioritization is statistically higher than non-prioritization. Nevertheless, we also observe that the performance of some warning-based prioritization (e.g., based on the warning amount of Flawfinder) can be lower than non-prioritization. We also check the performance with other K values (i.e., K = 30, 40, and 50% LOC) and the statistical analyses confirm that the performance of warning-based prioritization is better than non-prioritization.

> **Finding (warning-based prioritization)**: Compared to non-prioritization, warning-based prioritization using CodeQL warnings yields a substantial improvement for precision (12%) and Initial False Alarm (13%). CodeChecker offers a substantial improvement for recall (5.6%).

The different prioritization approaches can yield different performance gains. As seen in Table 6, prioritizing changed functions within VCCs with *Warning Density (WD)* usually yields the largest improvement for recall (up to 5.6%) and IFA (up to 13.3%) regardless of SASTs. In terms of precision, *WD* provides a slightly lower improvement than *Warning Amount (WA)* but *WA* normally offers a lower recall score and a higher IFA. On the other hand, *Warning Severity (WS)*, which developers typically use during the development process [68], yields the lowest performance improvement compared to other prioritization approaches across every metric.

> **Finding (prioritization approach)**: Prioritizing changed functions using *Warning Density (WD)* generally yields the largest improvement among the three approaches.

4.3 Waiting Time (RQ3)

Approach: To answer our RQ3: *How much computation time is required?*, we measure the computation time of each experiment i.e., executing a tool on one VCC. This period encompasses the entire process including the other necessary steps.

Results: Table 7 presents the computation time statistics. Flawfinder took the least computation time, with an average of 20 seconds, due to its syntactic analysis approach that bypasses compilation. On the other hand, CodeChecker, CodeQL, and Infer, which require compilation, took comparable average computation times ranging from 213 to 456 seconds. Surprisingly, Cppcheck, employing both semantic and syntactic techniques without compilation, took the longest computation time, with an average of 2,702 seconds.

> **Finding (average time)**: The average C/C++ SAST computation time is between 20 seconds and 45 minutes. Flawfinder is the fastest tool.

Computation time varies with system size. We measure system size (lines of code; LOC) in each VCC and examine computation times across VCCs. The average computation time on systems of different sizes reveals that Cppcheck is notably slower (over 10 minutes) for systems larger than 10k-15k LOC, while CodeQL, Infer, and CodeChecker are slower for systems larger than 50k-100k LOC. In contrast, Flawfinder's computation time remains relatively constant. The Cppcheck's prolonged computation time, especially for systems exceeding 10k-15k LOC, may be due to its sequential

[7] https://github.com/danbloomberg/leptonica/commit/cef50b2cb3a8be397d81d4d32388a3918374e1b5

Table 6: Percentage of performance differences when changed functions are ranked with warning-based prioritization in scenario S5 (*1Fn-Any*) at 25% of LOC in changed functions.

Tool	Prec@25% (WA) ↗	Prec@25% (WD) ↗	Prec@25% (WS) ↗	Recall@25% (WA) ↗	Recall@25% (WD) ↗	Recall@25% (WS) ↗	IFA (WA) ↘	IFA (WD) ↘	IFA (WS) ↘
CodeChecker	9.36%‡	8.09%‡	7.87%‡	**2.65%**	**5.66%**	**2.00%**	**-6.87%**	-10.29%†	**-5.81%**
CodeQL	**12.09%‡**	**10.98%‡**	**9.93%‡**	-4.72%	3.43%	-3.37%	-2.51%	**-13.34%‡**	-2.36%
Cppcheck	2.10%	2.10%	2.10%	0.81%	0.81%	0.81%	-0.44%	-0.39%	-0.44%
Flawfinder	9.87%†	2.51%	8.84%	-8.58%	-0.38%	-7.94%	7.85%	-4.23%	6.21%
Infer	9.04%‡	9.14%‡	9.05%‡	-1.98%	3.25%	-1.87%	-3.24%	-10.79%	-3.36%

‡ strongly significant: $p < 0.002$; † moderately significant: $0.002 \leq p < 0.01$ (α is adjusted with Bonferroni correction)

Table 7: Computation time statistics

Time (s)	CodeChecker	CodeQL	Cppcheck	Flawfinder	Infer
Miminum	3	26	1	**1**	2
Mean	213	343	2,702	**20**	456
Median	74	148	1,546	**6**	184
Maximum	7,268	3,314	16,974	**88**	2,305

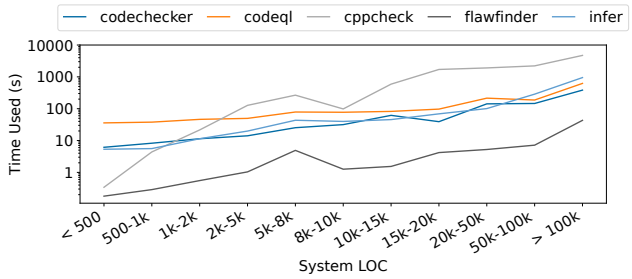


Figure 7: An average computation time across various system sizes in VCCs (Total LOC)

file analysis approach, resulting in bottlenecks with larger projects.

> **Finding (system size)**: The tools require longer computation time when the size of the system is larger than 50kLOC-100kLOC.

5 Discussion

5.1 SAST Performance for Code Review

Based on our empirical results, each SAST has shown promises and limitations for code review. Although Flawfinder has the highest detection rate (Figure 3) and takes the least computation time, warning-based prioritization using Flawfinder yields relatively low performance (Table 6). This is partly because Flawfinder produces a larger number of warnings to the changed functions than the other SASTs. This suggests that Flawfinder's abundance of warnings may not be ideal for prioritizing changed functions under limited review effort. On the other hand, while CodeQL and CodeChecker have lower detection rates compared to Flawfinder, their warnings effectively aid in prioritizing changed functions. When using CodeQL warnings for prioritization, the Initial False Alarm (IFA) and precision of finding vulnerable functions were also improved compared to non-prioritization. Similarly, CodeChecker can offer a substantial improvement in recall for warning-based prioritization. Our findings suggest that SASTs could assist secure code reviews by helping prioritize changed functions based on warnings. However, further refinement is necessary to support these practices better.

Although we analyze the warnings and vulnerability types based on CWE pillars in this study, the CWE item of a warning within the same CWE pillar as the VCC may not match the CWE item of the vulnerability in the corresponding CVE [48]. Thus, we examine whether the CWE item of the warning matches the CWE item of the vulnerability to understand **the accuracy of CWE pillar proxy**. We sample 400 warnings from the 7,564 warnings that all SASTs produced in vulnerable functions regardless of vulnerability types (*S5:1Fn-Any*), which allow us to draw a conclusion with a 95% confidence level and a 5% margin of error [38]. Following Li et al. [48]'s approach, we determine if the CWE item of a warning matches the CWE item of a VCC using the CWE item descriptions. We find that 66.42% of the warnings within the same CWE pillars as the VCC match the CWE items of the VCCs. We also find that 10.27% of the warnings with the different CWE pillars from the VCC actually match the CWE items of the VCCs.[8]

To provide more rigorous insights beyond the detection rate at the function level, we further investigate **the relevancy of the warnings** on the lines of code in VCCs (i.e., the warned lines) to the vulnerability reported in the CVEs. To do so, we manually analyze whether the warnings actually reflect the vulnerabilities in the VCCs. We examine the following information: the warned lines, the description of vulnerability in the associated CVE, and the fixing commit. We classify the warnings into three groups (following Thung et al. [65]): 1) **Relevant**–the warned line and the associated warning are relevant to the vulnerability and the fixing commit, 2) **Irrelevant**–the warned line and the warning are completely irrelevant to the vulnerability, and 3) **Unsure**–the warning is relevant to the vulnerability, but the warned line is not; and vice versa. The relevant group estimates a true positive rate of the warnings. The irrelevant group estimates the false positive rates of the warnings. We report the unsure group as it may still be beneficial in code reviews where the reviewers gain more security awareness from the adjacent warnings [33]. Based on the same sample set of 400 warnings, we find that 7% of the warnings are relevant to the warned line and vulnerability, 76% of the warnings are irrelevant to the warned line and vulnerability, and 17% of the warnings are either relevant to the warned line or vulnerability. These results,

[8]The evaluation results are included in the data package.

along with the findings in RQ2 (Section 4.2), suggest that warning information could aid reviewers in prioritizing which functions to review first. However, it remains essential for reviewers to invest meticulous effort in examining and identifying vulnerabilities within these prioritized functions.

5.2 Implications & Suggestions

We discuss the implications for practitioners, SAST developers, and researchers in this section.

5.2.1 For practitioners. Our results highlight the potential of using SASTs for secure code reviews. To maximize the benefits of SAST-supported secure code reviews, we provide the following recommendations. ① **SASTs can be leveraged to reduce code review effort.** Table 6 shows that at a fixed effort (25% LOC of changed functions), warning-based prioritization can increase the efficiency in identifying vulnerable functions compared to non-prioritization. RQ2 shows that prioritizing based on warning density generally provides a substantial improvement, suggesting that this prioritization approach could be adopted. Additionally, RQ3 shows that SAST's computing time is relatively short, which should fit within the code review waiting period [47]. ② **Changed functions that receive SASTs' warnings should be carefully investigated regardless of the warning types.** RQ1 shows that warning types may not be directly relevant to the vulnerability of the VCCs. Figure 3 shows that regardless of the warning types, SASTs can produce warnings in the vulnerable functions (S5:*1Fn-Any*); however, only 1%-46% of the VCCs receive warnings with the same type as their CVEs (S6:*1Fn-Same*). This result suggests that reviewers should not limit the focus of security review to the warning types that the tool produced. ③ **SAST should be carefully selected based on the project's security needs and resource constraints.** We found that several factors can impact the SASTs's effectiveness, e.g., vulnerability types, available time, and computation resources. Software projects should determine their security needs and resource constraints when choosing SASTs. If the matching vulnerability type is the main concern, specific security needs, such as types of vulnerability that lead to critical impact, should be recognized. For example, Table 5 shows that only CodeQL can detect vulnerability type **Access Ctrl** (CWE-284) and **Control Flow** (CWE-691), while only CodeChecker can detect vulnerability type **Cond Check** (CWE-703). Alternatively, when computing power is abundant, combining SASTs can increase the vulnerability detection effectiveness by 26% (Figure 4). If the project is exceptionally large, the computation time should be deliberately assessed as RQ3 indicates a substantial increase for projects exceeding 50kLOC-100kLOC. In addition, we find that setting up SASTs may demand technical expertise. Nevertheless, with the support of build automation tools, the setup of SASTs should be a one-time implementation effort.

5.2.2 For SAST's developers. Despite the potential values of SASTs, we find that incorporating them in secure code reviews still needs extra support. We offer the following recommendations to SAST's developers. ① **Provide vulnerability advice in warnings.** Our manual evaluation shows that some warnings are directly relevant to VCCs. However, some warnings with unmatching CWE pillars can also identify vulnerabilities. Figure 3 also shows that the VCCs with correct types of vulnerability can be reduced up to 35% at the function level, i.e., CodeQL in S5 (*1Fn-Any*) and S6 (*1Fn-Same*). Even if the relevant issue is detected, imprecise warning information can distract reviewers. SAST should highlight the risk if an issue can lead to vulnerability, otherwise the warning can be perceived as non-vulnerable. ②**Warning severity information should be improved.** Table 6 shows that prioritizing changed functions with warning severity does not yield a substantial improvement compared to other approaches. Since developers usually follow high-severity warnings during the development process [68], warning severity should be more reliable.

5.2.3 For researchers. Some practices in SAST-supported secure code reviews are not well understood. We suggest that future work explore the following aspects. ① **Rules for unconventional/unfinished code should be explored.** Code changes can be untidy during the development cycle and code reviews. In our RQ1, we observe that developmental code deviating from pre-defined rules can cause undetected vulnerabilities. AI models (e.g., Large Language Models) may help in detecting unprecedented vulnerabilities beyond the developer-defined rules of SASTs. However, a recent work [69] suggests that their vulnerability detection capability might be restricted by an insufficient semantic understanding of complex code. Therefore, future studies should investigate the solutions that detect vulnerabilities in such code. ② **Effective techniques to use SASTs for secure code review should be investigated.** Table 6 shows that warning-based prioritization with the three approaches can improve precision, recall, and IFA compared to non-prioritization. Meanwhile, certain approaches may cause performance drawbacks, e.g., prioritizing changed functions by warning amount of Flawfinder can reduce recall and increase IFA. Our manual evaluation also highlights the high false positive rates of the warnings (over 76%) in vulnerable changes, which can strongly hinder the effectiveness of code reviews [37]. These results and the findings in RQ2 (Section 4.2) suggest that warnings can assist reviewers in prioritizing functions, but reviewers must carefully examine and identify vulnerabilities within prioritized functions. Indeed, effective techniques to use SASTs for secure code review are still largely unexplored, which is an open challenge for future work. ③ **Undetected vulnerabilities should be prioritized.** Our result emphasizes that all SASTs can still miss many vulnerabilities in code changes by showing that 22% of VCCs do not receive the warnings in the vulnerable code changes from any tools (Section 4.1). The failure to detect real-world vulnerabilities stresses the importance of improving SASTs. Future work should explore the solutions to these shortcomings and minimize the undetected vulnerabilities. ④ **An ensemble approach of multiple SASTs should be investigated.** Figure 4 shows that combining tools can improve the effectiveness of vulnerability identification. However, it is infeasible for projects with limited computing resources to use many SASTs, especially for large projects (Figure 7). A practical guideline to combine SASTs is helpful for practitioners.

6 Threats to Validity

We discuss potential threats to this study's validity in this section.

Internal validity: The VCC dataset quality impacts study results. We verified vulnerable changes, discarded VCCs that could not be

linked to fixing commits, and excluded a redundant commit caused by a renamed project. For SAST selection, we carefully chose the popular SASTs that practitioners can employ in the development process. We were aware that enabling different SAST's rules can affect the effectiveness in general. We adhered to default rules because tool developers strongly recommend it. In addition, we encountered out-of-memory issues when attempting to enable extra rules. Upon manual mapping of the warnings to the CWEs pillar (Section 3.3.3), human biases can potentially occur. To address this, the two authors first lay the groundwork for mapping before proceeding collaboratively with the remaining warnings. Similarly, we acknowledge the potential error during the manual evaluation of the warnings. Lastly, as MITRE regularly updates CWE and CVE entries, the sustainability of the results cannot be guaranteed.

Construct validity: The construct validity concerns the measured indicators. Based on the previous literature [53, 66, 68] , we assume that reviewers prioritize the changed functions with the warnings during code reviews. We analyzed the prioritization performance at various fixed efforts (K% LOC) to ensure consistency. To increase the validity, we explore various prioritization approaches, i.e., warning amount, density, and severity. However, reviewers may adopt other approaches when reviewing code changes, introducing potential variations beyond the examined constructs.

External validity: The external validity concerns the generalizability of the results. Our findings are supported by the breadth of our dataset, which contains 815 commits from 92 diverse projects that contributed to 319 vulnerabilities. Our results are based on five SASTs. A broader set of SASTs may improve the generalizability of the findings. Nonetheless, these five SASTs are commonly used, actively maintained, and compatible with the studied systems and executable via CLI (Section 3.3.1). To support future work, we release a framework [35] that future work can extend for additional SASTs and code change datasets in diverse languages. We also acknowledge that the findings regarding the relevancy of the warning (Section 5.1) are based on the sample warnings, which may have limited generalizability. For transparency and to facilitate future work, we release the benchmark framework, datasets, and evaluation results.

7 Related Work

The effectiveness of SASTs on VCCs has not been previously investigated. Most existing works primarily focused on evaluating the SASTs on the released versions of software or the crafted programs that contain known vulnerabilities. The datasets of code commits submitted by developers for code reviews are not usually used by SAST studies. Lipp et al. [49] evaluates five free and one commercial C/C++ SASTs and reported that SASTs can miss 47%-80% of vulnerabilities. Their benchmark dataset includes 111 front-ported vulnerabilities from Magma dataset [42] and 81 vulnerabilities from released versions of Binutils and FFmpeg. In this work, we used the dataset of vulnerability contributing commits which include vulnerable functions and vulnerable files, and found that SASTs can detect at least one vulnerable function of 78% of VCCs. Kannavara [46] evaluated the effectiveness of *Klocwork*, a multi-language SAST, on the Linux Kernel. However, only one version of the Linux Kernel was used. In this work, we investigate multiple versions of code (815 VCCs) from the development process. A recent study by Li et al. [48] investigated a closely relevant context to our work. They evaluated seven SASTs using code commits and reported that SASTs detect only 12.7% of the vulnerabilities with the correct vulnerability types. However, they focused on Java SASTs and seven types of CWE vulnerability. In this work, we study C/C++ SASTs using the VCC dataset with eight types of CWE vulnerability.

Differing from previous works, we focus on real-world developmental code commits that are the subject of code reviews. This novel dataset can enhance the understanding of SAST's performance, offering insight into SAST's performance in the development phase of a software project. We are also the first to investigate warning-based prioritization and show that the effectiveness in identifying vulnerabilities can be improved when the changed functions are ranked with the information from SAST warnings.

8 Conclusion

In this study, we conducted empirical research to understand the practical benefits of C and C++ SASTs for secure code reviews. The results show that 52% of VCCs can be warned by a single tool in the changed functions that contain vulnerable code. By combining tools, the detection effectiveness can increase by 26%. For secure code reviews, prioritizing changed functions with SAST warnings can improve accuracy (i.e., 12% of precision and 5.6% of recall) and reduce Initial False Alarm (i.e., lines of code in non-vulnerable functions that must be inspected until the first vulnerable function) by 13%. Moreover, the average computing time of SASTs should fit within the waiting time of the code reviews. However, at least 76% of the warnings in vulnerable functions can be irrelevant to the vulnerability in VCCs, and 22% of the VCCs can be undetected.

Based on these findings, we suggest software projects select SASTs that fit their security needs and resource constraints, leverage SASTs to streamline code review efforts, and inspect changed functions with SAST warnings, irrespective of warning types. On the other hand, SAST's developers should enhance vulnerability advice in warnings and improve warning severity information. Future work can improve SASTs on the undetected vulnerabilities, explore SAST rules for developmental code, investigate effective techniques for using SASTs in secure code reviews, and explore an ensemble approach involving multiple SASTs. We also release the SAST evaluation framework [35] and dataset to support forthcoming investigations.

Data Availability Statement

We have released the datasets and scripts used in this study to foster future works on SAST-supported secure code reviews [36].

Acknowledgment

This work was supported by the use of Nectar Research Cloud, a collaborative research platform supported by NCRIS-funded Australian Research Data Commons (ARDC). Patanamon Thongtanunam and Van-Thuan Pham were supported by two Australian Research Council's Discovery Early Career Researcher Award (DECRA) projects (DE210101091 and DE230100473). We sincerely thank anonymous reviewers for their valuable feedback.

References

[1] [n. d.]. About comparing branches in pull requests - GitHub Docs. https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-comparing-branches-in-pull-requests/

[2] [n. d.]. CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container - SEI CERT C++ Coding Standard - Confluence. https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR51-CPP.+Use+valid+references%2C+pointers%2C+and+iterators+to+reference+elements+of+a+container

[3] [n. d.]. CVE | Overview. https://www.cve.org/About/Overview

[4] 2015. NVD - CVE-2015-8895. https://nvd.nist.gov/vuln/detail/CVE-2015-8895

[5] 2015. NVD - CVE-2015-9059. https://nvd.nist.gov/vuln/detail/CVE-2015-9059

[6] 2017. NVD - CVE-2017-12997. https://nvd.nist.gov/vuln/detail/CVE-2017-12997

[7] 2017. NVD - CVE-2017-13040. https://nvd.nist.gov/vuln/detail/CVE-2017-13040

[8] 2017. NVD - CVE-2017-13083. https://nvd.nist.gov/vuln/detail/CVE-2017-13083

[9] 2018. NVD - CVE-2018-17294. https://nvd.nist.gov/vuln/detail/CVE-2018-17294

[10] 2018. NVD - CVE-2018-7186. https://nvd.nist.gov/vuln/detail/CVE-2018-7186

[11] 2018. NVD - CVE-2018-7485. https://nvd.nist.gov/vuln/detail/CVE-2018-7485

[12] 2020. NVD - CVE-2020-5260. https://nvd.nist.gov/vuln/detail/CVE-2020-5260

[13] 2021. Standard LGTM queries for C/C++. https://github.com/github/codeql/blob/main/cpp/ql/src/codeql-suites/cpp-lgtm.qls

[14] 2022. Clang-Tidy Checkers. https://releases.llvm.org/13.0.1/tools/clang/tools/extra/docs/clang-tidy/checks/list.html

[15] 2023. CWE - CWE-457: Use of Uninitialized Variable (4.12). https://cwe.mitre.org/data/definitions/457.html

[16] 2023. CWE - CWE-664: Improper Control of a Resource Through its Lifetime (4.12). https://cwe.mitre.org/data/definitions/664.html

[17] 2023. CWE - CWE-699: Software Development (4.12). https://cwe.mitre.org/data/definitions/699.html

[18] 2023. CWE - CWE-825: Expired Pointer Dereference (4.12). https://cwe.mitre.org/data/definitions/825.html

[19] 2023. Flawfinder. https://dwheeler.com/flawfinder/

[20] 2023. Infer Static Analyzer. https://fbinfer.com/

[21] 2023. Source Code Security Analyzers - NIST. https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers

[22] 2024. Clang Checkers. https://clang.llvm.org/docs/analyzer/checkers.html

[23] 2024. CodeChecker. https://codechecker.readthedocs.io/en/latest/

[24] 2024. CodeQL. https://codeql.github.com/

[25] 2024. Cppcheck - A tool for static C/C++ code analysis. https://cppcheck.sourceforge.io/

[26] 2024. Infer - Issue types. https://fbinfer.com/docs/all-issue-types/

[27] 2024. terryyin/lizard: A simple code complexity analyser without caring about the C/C++ header files or Java imports, supports most of the popular languages. https://github.com/terryyin/lizard

[28] Bushra Aloraini and Meiyappan Nagappan. 2017. Evaluating state-of-the-art free and open source static analysis tools against buffer errors in android apps. In *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*. Institute of Electrical and Electronics Engineers Inc., 295–306. https://doi.org/10.1109/ICSME.2017.77

[29] Bushra Aloraini, Meiyappan Nagappan, Daniel M. German, Shinpei Hayashi, and Yoshiki Higo. 2019. An empirical study of security warnings from static application security testing tools. *Journal of Systems and Software* 158 (12 2019), 110427. https://doi.org/10.1016/J.JSS.2019.110427

[30] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings - International Conference on Software Engineering*. 931–940. https://doi.org/10.1109/ICSE.2013.6606642

[31] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern code reviews in open-source projects: Which problems do they fix?. In *11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings*. Association for Computing Machinery, 202–211. https://doi.org/10.1145/2597073.2597082

[32] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016* 1 (5 2016), 470–481. https://doi.org/10.1109/SANER.2016.105

[33] Larissa Braz and Alberto Bacchelli. 2022. Software Security during Modern Code Review: The Developer's Perspective. 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, 810–821. https://doi.org/10.1145/3540250.3549135

[34] Larissa Braz, Enrico Fregnan, Gul Calikli, and Alberto Bacchelli. 2021. Why don't developers detect improper input validation? '; DROP TABLE Papers; -. In *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, 499–511. https://doi.org/10.1109/ICSE43902.2021.00054

[35] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. 2024. An Extensible Automated Benchmarking Framework for Static Analysis Tool Evaluation Using Open-Source Code Commits. https://doi.org/10.5281/zenodo.10259921

[36] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. 2024. Dataset for An Empirical Study of Static Analysis Tools for Secure Code Review. https://doi.org/10.5281/zenodo.12653656

[37] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery (ACM), 332–343. https://doi.org/10.1145/2970276.2970347

[38] Marco Di Biase, Magiel Bruntink, and Alberto Bacchelli. 2016. A security perspective on code review: The case of chromium. In *Proceedings - 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation, SCAM 2016*. Institute of Electrical and Electronics Engineers Inc., 21–30. https://doi.org/10.1109/SCAM.2016.30

[39] Michael D. Ernst. 2004. Static and dynamic analysis: synergy and duality. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. Association for Computing Machinery (ACM), 35–35. https://doi.org/10.1145/996821.996823

[40] Katerina Goseva-Popstojanova and Andrei Perhinschi. 2015. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology* 68 (12 2015), 18–33. https://doi.org/10.1016/J.INFSOF.2015.08.002

[41] Mark Harman and Peter O'Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *Proceedings - 18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018*. Institute of Electrical and Electronics Engineers Inc., 1–23. https://doi.org/10.1109/SCAM.2018.00009

[42] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, Vol. 4. Association for Computing Machinery (ACM), 1–29. https://doi.org/10.1145/3428334

[43] Sarah Heckman and Laurie Williams. 2008. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *ESEM'08: Proceedings of the 2008 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. 41–50. https://doi.org/10.1145/1414004.1414013

[44] Yang Hong, Chakkrit Kla Tantithamthavorn, and Patanamon Pick Thongtanunam. 2022. Where Should i Look at? Recommending Lines that Reviewers Should Pay Attention to. In *Proceedings - 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022*. Institute of Electrical and Electronics Engineers Inc., 1034–1045. https://doi.org/10.1109/SANER53432.2022.00121

[45] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2022. The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study. *IEEE Transactions on Software Engineering* (2022). https://doi.org/10.1109/TSE.2022.3140868

[46] Raghudeep Kannavara. 2012. Securing opensource code via static analysis. In *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*. 429–436. https://doi.org/10.1109/ICST.2012.123

[47] Gunnar Kudrjavets, Aditya Kumar, Nachiappan Nagappan, and Ayushi Rastogi. 2022. Mining Code Review Data to Understand Waiting Times Between Acceptance and Merging: An Empirical Analysis. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, 579–590. https://doi.org/10.1145/3524842.3528432

[48] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. 2023. Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*. Association for Computing Machinery. https://doi.org/10.1145/3611643.3616262

[49] 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, New York, NY, USA, 544–555. https://doi.org/10.1145/3533767.3534380

[50] Mika V. Mäntylä and Casper Lassenius. 2009. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering* 35, 3 (2009), 430–448. https://doi.org/10.1109/TSE.2008.71

[51] Sahar Mehrpour and Thomas D. LaToza. 2022. Can static analysis tools find more defects? *Empirical Software Engineering* 28, 1 (2 2022). https://doi.org/10.1007/S10664-022-10232-4

[52] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An open fuzzer benchmarking platform and service. In *ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Vol. 21. Association for Computing Machinery, Inc, 1393–1403. https://doi.org/10.1145/3468264.3473932

[53] Tukaram Muske and Alexander Serebrenik. 2016. Survey of approaches for handling static analysis alarms. In *Proceedings - 2016 IEEE 16th International Working*

Conference on Source Code Analysis and Manipulation, SCAM 2016. Institute of Electrical and Electronics Engineers Inc., 157–166. https://doi.org/10.1109/SCAM.2016.25

[54] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. 2022. A large-scale study of usability criteria addressed by static analysis tools. In *ISSTA 2022 - Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, Inc, 532–543. https://doi.org/10.1145/3533767.3534374

[55] Paulo Nunes, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. 2017. On Combining Diverse Static Analysis Tools for Web Security: An Empirical Study. In *Proceedings - 2017 13th European Dependable Computing Conference, EDCC 2017*. Institute of Electrical and Electronics Engineers Inc., 121–128. https://doi.org/10.1109/EDCC.2017.16

[56] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2015. Would static analysis tools help developers with code reviews?. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., 161–170. https://doi.org/10.1109/SANER.2015.7081826

[57] Massimiliano Di Penta, Luigi Cerulo, and Lerina Aversano. 2009. The life and death of statically detected vulnerabilities: An empirical study. *Information and Software Technology* 51, 10 (10 2009), 1469–1484. https://doi.org/10.1016/J.INFSOF.2009.04.013

[58] Peter C Rigby and Christian Bird. 2013. Convergent Contemporary Software Peer Review Practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. ACM Press, New York, New York, USA. https://doi.org/10.1145/2491411

[59] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating SZZ implementations through a developer-informed oracle. In *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, 436–447. https://doi.org/10.1109/ICSE43902.2021.00049

[60] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *Proceedings - International Conference on Software Engineering*, Vol. 1. IEEE Computer Society, 598–608. https://doi.org/10.1109/ICSE.2015.76

[61] Devarshi Singh, Varun Ramachandra Sekar, Kathryn T. Stolee, and Brittany Johnson. 2017. Evaluating how static analysis tools can reduce code review effort. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, Vol. 2017-October. IEEE Computer Society, 101–105. https://doi.org/10.1109/VLHCC.2017.8103456

[62] ŚliwerskiJacek, ZimmermannThomas, and ZellerAndreas. 2005. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes* 30, 4 (5 2005), 1–5. https://doi.org/10.1145/1082983.1083147

[63] Darko Stefanović, Danilo Nikolić, Dušanka Dakić, Ivana Spasojević, and Sonja Ristić. 2020. Static code analysis tools: A systematic literature review. In *Annals of DAAAM and Proceedings of the International DAAAM Symposium*, Vol. 31. DAAAM International Vienna, 565–573. https://doi.org/10.2507/31ST.DAAAM.PROCEEDINGS.078

[64] Christopher Thompson and David Wagner. 2017. A large-scale study of modern code review and security in open source projects. In *ACM International Conference Proceeding Series*. Association for Computing Machinery, 83–92. https://doi.org/10.1145/3127005.3127014

[65] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. 2012. To what extent could we detect field defects? An empirical study of false negatives in static bug finding tools. In *2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings*. 50–59. https://doi.org/10.1145/2351676.2351685

[66] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. 2023. Are automated static analysis tools worth it? An investigation into relative warning density and external software quality on the example of Apache open source projects. *Empirical Software Engineering* 28, 3 (6 2023), 1–21. https://doi.org/10.1007/S10664-023-10301-2

[67] Stephen Turner. 2014. Security vulnerabilities of the top ten programming languages: C, Java, C++, Objective-C, C#, PHP, Visual Basic, Python, Perl, and Ruby. *Journal of Technology Research* (2014), 1–16. http://gauss.ececs.uc.edu/Courses/c6056/pdf/131731.pdf

[68] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25, 2 (3 2020), 1419–1457. https://doi.org/10.1007/S10664-019-09750-5

[69] Zhilong Wang, Lan Zhang, Chen Cao, and Peng Liu. 2023. The Effectiveness of Large Language Models (ChatGPT and CodeBERT) for Security-Oriented Code Analysis. (7 2023). https://doi.org/10.48550/arXiv.2307.12488

[70] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2022. Predicting Defective Lines Using a Model-Agnostic Technique. *IEEE Transactions on Software Engineering* 48, 5 (5 2022), 1480–1496. https://doi.org/10.1109/TSE.2020.3023177

[71] Charles Weir, Sammy Migues, and Laurie Williams. 2022. Exploring the Shift in Security Responsibility. *IEEE Security and Privacy* 20, 6 (2022), 8–17. https://doi.org/10.1109/MSEC.2022.3150238

Received 2024-04-12; accepted 2024-07-03