How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage

Nasif Imtiaz[1], Brendan Murphy[2], and Laurie Williams[1]

[1]Department of Computer Science, North Carolina State University
[2]Microsoft Research, Cambridge, UK
simtiaz@ncsu.edu, bmurphy@microsoft.com , lawilli3@ncsu.edu

***Abstract*—Static analysis tools (SATs) often fall short of developer satisfaction despite their many benefits. An understanding of how developers in the real-world act on the alerts detected by SATs can help improve the utility of these tools and determine future research directions. The goal of this paper is *to aid researchers and tool makers in improving the utility of static analysis tools through an empirical study of developer action on the alerts detected by Coverity, a state-of-the-art static analysis tool.* In this paper, we analyze five open source projects as case studies (Linux, Firefox, Samba, Kodi, and Ovirt-engine) that have been actively using Coverity over a period of at least five years. We investigate the alert occurrences and developer triage of the alerts from the Coverity database; identify the alerts that were fixed through code changes (i.e. actionable) by mining the commit history of the projects; analyze the time an alert remain in the code base (i.e. lifespan) and the complexity of code changes (i.e. fix complexity) in fixing the alert. We find that 27.4% to 49.5% (median: 36.7%) of the alerts are actionable across projects, a rate higher than previously reported. We also find that the fixes of Coverity alerts are generally low in complexity (2 to 7 lines of code changes in the affected file, median: 4). However, developers still take from 36 to 245 days (median: 96) to fix these alerts. Finally, our data suggest that severity and fix complexity may correlate with an alert's lifespan in some of the projects.**

***Index Terms*—static analysis, tools, alerts, warnings, developer action**

I. Introduction

Modern software projects, both commercial and open source, are increasingly integrating static analysis tools (SATs) as a part of their development process [14], [19], [40]. Despite the benefits of SATs in helping higher software quality [33], their shortcomings are well-known and well-studied in the literature [19], [33]. Unactionable, untrustworthy, and incomprehensible alerts detected by SATs are some of the major complaints against these tools [41]. While prior work has investigated the actionability rate (the rate of true positive alerts) of SATs [21], [34], [36], there is little empirical evidence on how developers in the real-world act upon SAT alerts when they are *actively* using the tool.

Previously, Guo et al. has studied which Coverity [42] alerts are likely to be inspected by Linux developers [26]. In recent years, Marcilo et al. [21] and Zampetti et al. [45] have studied Java projects that use SATs. Their research investigated the actionability of the alerts and the time involved in fixing them. We build upon this prior work by empirically analyzing five large open source projects as case studies (Linux, Firefox, Samba, Kodi, and Ovirt-engine) that have been actively using Coverity, a state-of-the-art commercial SAT, for at least past five years. The maintainers of these projects confirmed that they regularly monitor alerts detected by Coverity. We collect alert history and developer triage of the alerts from the Coverity defect database that is private to the project developers. We then map the detection history of the eliminated alerts with the commit history of the affected files to determine if the alert was fixed by developers making code changes.

Specifically, we want to know the actionability scenario of SATs within real development context, that is, the alerts inducing a subsequent developer action in the code. An empirical evidence of actionability of SATs will inform us where we currently stand. Furthermore, how long developers take to fix the alerts; the effort involved in the fix; and what factors affect the time an alert will remain in the code base will determine future research directions in prioritizing the alerts and motivating the developers to the use of SATs. We conduct our research on the usage of Coverity. The goal of this paper is *to aid researchers and tool makers in improving the utility of static analysis tools through an empirical study of developer action on the alerts detected by Coverity, a state-of-the-art static analysis tool.* We begin with asking:

> ***RQ 1:*** (Occurrence) To what extent do Coverity alerts occur in terms of alert type?
>
> ***RQ 2:*** (Actionability) Which Coverity alerts are fixed by the developers through code change?
>
> ***RQ 3:*** (Lifespan) How long do Coverity alerts remain in the code before being fixed?
>
> ***RQ 4:*** (Fix complexity) What is the complexity of the code changes that fix a Coverity alert?

Next, we hypothesize that developers will pay attention to alerts of higher severity as marked by the tool faster than the alerts lower in severity as severity indicates the impact or risk of the potential defect pointed out by the alert:

> ***RQ 5:*** Do Coverity alerts with higher severity have shorter lifespan?

We also hypothesize that the alerts that are easier to fix will get fixed quicker than the alerts with a difficult fix:

RQ 6: Do Coverity alerts with lower fix complexity have shorter lifespan?

In summary, **our contributions are:**

1) An empirical study of developer action on static analysis alerts based on the active usage of Coverity, a state-of-the-art static analysis tool, by five large open source projects;
2) An empirical analysis of complexity of code changes to fix Coverity alerts; and
3) An empirical analysis of the correlation between fix complexity and severity of Coverity alerts with the time the alerts remain in the main code base.

II. Background & Related Work

A. Static Analysis Tools: Pros & Cons

SATs detect potential code defects without having to execute the code [33], and therefore, can be "shifted-left" [11] to find and prevent defects early in the development process. SATs detect several types of defects, including *security vulnerabilities, performance issues, quality issues, and bad programming practices (code smells)* [36]. For example, the use of static analysis could have detected vulnerabilities that have been exploited [38], [40], [44]. The literature denotes the potential defects detected by SATs as *alerts, warnings, violations, or issues* [36]. In this paper, we refer to them as **alerts**.

SATs are increasingly becoming an integral part of modern software development [14], [19], [40]. Large software companies have started building program analysis ecosystems (e.g. Google's Tricorder [41], Microsoft's CloudBuild [24]) that may integrate multiple static analyzers to its pipeline. Open source projects also use static analysis in various degrees. Prior work found that half of the state-of-the-art open source (OSS) projects have employed SATs [14]. Several commercial [6], [9], [42] and open source [2], [4], [16] static analysis tools are available for developers to use.

However, SATs come with several shortcomings which can make developers turn away from using the tools [33]. SATs can flood the developers with large volume of alerts that includes a high rate of unactionable ones which may result in developers grow a habit of ignoring the SATs altogether [28], [33]. An **unactionable** alert may be one of the following [36]:

1) a trivial concern to fix;
2) less likely to manifest in runtime environment; or
3) incorrectly identified due to the limitations of the tool.

Prior research has looked at developer perception towards SATs. Christakis et al. [19] surveyed developers at Microsoft to identify barriers to use SATs and the characteristics developers expect from these tools. The researchers found that unnecessary rule checkers that are on by default, bad warning messages, and high false positive rate make SATs difficult to use while the ability to suppress warnings is an important need for the developers. These finding align with Imtiaz et al.'s [30] study of Stack Overflow posts where they find that developers' most frequent questions involve how to ignore alerts based on different criteria (e.g. alert type, code location etc.); and validation of suspected false positive alerts. Sadowski et al. [40] performed a similar study in Google and found unactionable, incomprehensible, and untrustworthy alerts are reasons that developers do not want to use SATs. Research has also shown that these challenges are generic to all static analysis tools [30], [32].

B. Actionability of Static Analysis Tools

Prior research has investigated the actionability rate of SATs. A common methodology to this work is for the researchers to run SATs with the default configuration over a sequence of historical revisions of projects to identify the alerts that come and go between code changes [27], [34], [36]. Through this method, Liu et al. [36] ran FindBugs [4] on 730 Java projects to find a small fraction of alerts (less than 1%) to be fixed by the developers. Kim et al. [34] ran three SATs on three Java programs and found that only 6% to 9% of the alerts are removed by bug fix changes. Heckman et al. [27] has run FindBugs on six small to medium scale Java projects and manually identified the true alerts to find true positive rate to range between 8.2% to 50.0%. However, the threat to this methodology in understanding developer action is that developers are not actively using the tool and monitoring the alerts [21]. Additionally, developers can tune the tool to project specific needs [43] to reduce false positives, or disable certain alert types that they do not deem as important. Tuning increases the probability of detecting actionable alerts for the tools as the tool is then customized to the specific project.

Toolmakers and researchers have published their experience of running SATs in the "real world" [13], [15]. However, to the best our knowledge, few empirical studies have been conducted on developer action to alerts when the developers were actively using the tool. In recent work, Marcilio et al. [21] has studied 246 open source projects and 2 government projects (all Java projects) where developers use SonarQube [7]. Based on the data on SonarQube defect database, they find developers only fix 13% of the alerts on average and take 18.99 days (median) to fix them. Zampetti et al. [45] has studied 20 Java open source projects that use SATs within Travis CI [8] pipeline to investigate how often alerts break builds, the type of the alerts, how long developers take to fix them, and how do they fix them (e.g. change in source code, or build scripts, or tool configuration).

Our work differs from [21], [31], [45] in the methodology of determining actionability. We map alert detection history with the commit history of the affected files to determine if the alerts are fixed through code changes as we observed in our study that tools can stop detecting alert (and therefore, mark as fixed) due to multiple reasons other than developer fix (e.g. change in tool configuration, tool update, suppress alert through source code annotation, deletion of file). Our case studies include both C/C++ and Java to determine if developer action on SATs are generic to both languages. Furthermore, our study also presents an empirical study on the complexity of code changes when developers are fixing the SAT alerts.

III. Case Studies: Tool & Projects

In this paper, we analyze five projects that use Coverity. We collect data from Coverity defect database and respective project's source code repository. In this section, we briefly explain the tools, selected projects, and the data source.

A. Coverity Scan

Coverity [42], developed by Synopsys, is a commercial SAT and is considered to be one of the best SATs with high fix rates and low false positives [19], [22]. Synopsys offers a free cloud-based service for open source projects. The service is named as 'Coverity Scan' [3]. As per the 2017 annual report, Coverity Scan hosts 4,600 active projects and claims to have detected 1.1 million defects (as alerts) [37]. Coverity Scan also maintains a defect database on its own cloud server for each project that it hosts. The defect database contains information regarding all the alerts and their respective triage history. In this paper, we will refer to Coverity Scan simply by the name of the tool, Coverity.

Coverity supports multiple languages including C/C++ and Java. Developers have an option to **triage**[1] the alerts on Coverity's defect database. Developers can manually triage information on alert attributes such as classification, action, and assignee which are initially set as null by the tool when a new alert is detected. Developers can triage an alert as a 'False Positive' (incorrectly identified due to tool limitation) or as 'Intentional' (the alert is correct, but the code is written intentionally by the developer), in which case Coverity will stop reporting that alert. Conversely, developers can also triage an alert as a 'Bug'. However, developers may not always triage alerts on Coverity's defect database, even if they make a code change to fix the alert or in case of false positives.

Additionally, Coverity provides four tuning features which can be used to reduce the detection of unactionable alerts based on project-specific requirements:

- Modeling file: Developers can write modeling files to overcome false positives that is caused by the absence of source code for a function during the build [43].
- Code annotations with suppression keywords: Developers can add suppression keywords to source code to ignore certain types of alerts in a code location.
- Ignore specific files: Developers can instruct Coverity to ignore specific files or directories during analysis.
- Disable checker: Coverity comes with a set of rule checkers (each rule checker identifies a certain type of alert in the code) for each language that are enabled by default. Developers have the option to disable a checker.

B. Coverity Defect Database

Coverity maintains a defect database for each of the projects it hosts. The database contains historical analysis reports along with the time of the analysis. Aggregating all the analysis reports, we get the list of all distinct generated alerts. For each alert in the database, we have information on their detection history, alert type, and impact. Alert type is a categorization of alerts provided by Coverity based on the rule checker that detects the alert (e.g. Logically dead code). Each alert type is assigned an impact level of 'Low', 'Medium', and 'High' by the tool based on the risk severity. When developers triage an alert to classify as 'False Positive' or 'Intentional', Coverity will stop reporting that alert and update its status to 'Dismissed'. Otherwise, when Coverity does not detect an alert anymore when analyzing a new version of the code, the alert's status is changed to 'Fixed'. However, we will refer to the 'Fixed' alerts by the term ***'eliminated'*** as alert can go away either through developer fix or through other possible ways (e.g. file deletion, tool update, change in the tool configuration).

When Coverity detects a new alert, the tool may also present a detailed event history on how the alert is manifested along with file name and line location for each associated event. However, in the data that we collected, information on detailed event history and exact line location are missing for the majority of the eliminated alerts. Therefore, we could only use the file location of an alert in order to determine if the alert was fixed through code change.

C. Projects

We analyzed four large open source projects that are written in C/C++ and use Coverity's C/C++ checkers. We also analyzed one large Java project to investigate if our results are applicable beyond C/C++. We select these five projects based on following criteria:

- The projects have used Coverity for at least five years with a median interval of analysis frequency of less than one week (therefore, having at least 260 historical analysis reports);
- The latest analyzed version of project contains at least 100,000 lines of code, excluding blank lines and comment lines (ensuring large projects);
- The project maintainers have granted the first author observation access; and
- Developers of these projects have confirmed that they monitor Coverity reports and how the projects are configured on Coverity (repository locations and analyzed branches).

Table I lists the details of our five selected projects. The projects come from a variety of domains: Linux is an operating system; Firefox is a web-browser; Samba is a file/printer sharing system; Kodi is a home theater and entertainment hub; and Ovirt-engine is a server and desktop virtualization platform.

To ensure that the developers from these projects monitors Coverity alerts, we reached out to the project maintainers through respective project's developer forum or through emailing the developers who are listed as admin on respective project's Coverity defect database. In our emails and forum

[1]Triage is a process to screen and prioritize alerts. Coverity uses the term 'triage' when developers manually update any information regarding an alert.

TABLE I
ANALYZED PROJECTS

Project	Language	Analysis Reports	Start Date	End Date	Interval (days)	Analyzed Lines of Code
Linux	C/C++	598	2012-05-17	2019-04-08	3	9,335,805
Firefox	C/C++	662	2006-02-22	2018-10-26	2	5,057,890
Samba	C/C++	714	2006-02-22	2019-01-02	3	2,136,565
Kodi	C/C++	394	2012-08-28	2019-03-19	3	388,929
Ovirt-engine	Java	790	2013-06-24	2019-03-18	1	409,018

TABLE II
HOW PROJECTS MONITOR AND CONFIGURE COVERITY

	Monitoring Alert			Configuring Coverity to Reduce Unactionable Alerts			
Project	Audit alert	Triage alerts	Triage Rate	Update Modelling file	Use code annotations	Ignore files	Disable checker
Linux	Sometimes	Sometimes	15.2%	Sometimes	Never	Never	Never
Firefox	Always	Most of the time	36.3%	Sometimes	Sometimes	Sometimes	Never
Samba	"Regularly monitor and act on it"		54.4%		No Information		
Kodi	No Information		43.2%		No Information		
Ovirt-engine	Most of the time	Always	56.5%	Never	Never	Never	Never

posts, we also added a survey[2] that asks: 1) how frequently the developers monitor Coverity reports; and 2) how frequently the developers use the tuning features that Coverity provides. We used a 5-level Likert scale ranging from Always to Never in the survey to estimate the frequency. Developers from Linux, Firefox, and Ovirt-engine completed our survey confirming they monitor Coverity alerts. While developers from Samba and Kodi did not complete our survey, in reply to our post on their developer forums, they have confirmed us that they monitor Coverity alerts. Another means of confirming that the developers monitor Coverity alerts is to look at if the alerts get triaged by the developers on Coverity defect database. We see the portion of alerts that get triaged (Triage Rate) on Coverity defect database vary between 15.2% to 56.5%. Table II summarizes how the five projects monitor Coverity alerts based on developer response and triage rate. Maintainers of these projects also confirmed that Coverity analyzes the 'master' branch in the database we have access to.

D. Mining Project Repositories

Modern software projects use version control systems (e.g. Git [5]) that keep historical records of every code change and associated information. We mined the Git repositories of the five projects to get the commit history of the project files. We extracted the full commit history of each file, even if the file was deleted or renamed at any point. For each commit, we extract author name; author date; committer name; commit date; affected files; and the commit diffs (changes between commits). Furthermore, we determine the date and time when a commit was merged into the 'master' branch using an open source tool [3]. Out of the five analyzed projects, Firefox uses Mercurial as its version control system, while the other projects use Git. However, we used a mirrored Git repository of Firefox to mine commit history as we analyze every project through the same analysis scripts.

[2]https://go.ncsu.edu/llgnnrw

[3]https://github.com/mhagger/git-when-merged

IV. DATA COLLECTION & PREPROCESSING

A. Step 1: Collect Coverity Data

We begin with collecting alert history data from the Coverity defect database. For each alert, we use the following attributes:

- **CID** : A unique identifier for each distinct alert;
- **Type**: Coverity reports the type of each alert based on the rule checker that detected it;
- **Impact**: Severity of the alert (High, Medium, Low);
- **First Detected**: The date an alert was first detected;
- **Status**: New for alerts that remained in the code at the time of the last analysis report collected; Fixed for eliminated alerts; and Dismissed for alerts marked as false positive or intentional by the developers;
- **Classification**: The alerts are initially set as 'Unclassified' by default. Developers may triage some alerts to manually classify the alert as Bug; False Positive; Intentional; Pending.
- **File**: The full file path of the file where the alert is detected; and
- **Last Snapshot ID**: the unique identifier of the last analysis report an alert was reported by Coverity.

We also use the following attributes of each analysis report from Coverity defect database in our analysis:

- **Snapshot ID**: A unique identifier for each analysis report within each project;
- **Date**: The timestamp of when the analysis was run; and
- **Code Version Date**: The timestamp of the source code version analyzed in the report. This time is nearly identical to the 'Date' field.

B. Step 2: Data Preprocessing

We collected information for 89,125 alerts from Coverity defect database in Step 1. We then performed a series of data preprocessing steps:

(a) In our data, some alerts' first detection date were earlier than the date of the first analysis report that we collected. We discarded these alerts from our dataset as we do not

have the history of the eliminated or dismissed alerts before the first report, but only the alerts that were still getting detected at the time of the first report. We discarded 6,091 alerts through this filtering process.

(b) When a file is renamed or moved to a different location, Coverity assumes that alert as 'Fixed' and creates a new identifier for the same alert detected on the new file path. As a result, we get duplicates in our data. By analyzing commit history, we detected if an alert went away due to file renaming (within the time when an alert was last detected and first eliminated, there is a commit on the file that changes its full file path to a new one). If so, we discarded that alert from our data set. We also investigated if a new alert of the same type was generated on the new file location at the time when the alert in the old file location was eliminated. If yes, we determine the two alerts as one and update their first and last detection time. In this way, we identified 252 alerts as duplicates.

(c) Unactionable alerts generated by SATs is a part of our study. However, we noticed four cases where an unusually large number of alerts of the same type were detected at the same time and were eventually eliminated at the same time in a subsequent analysis report without any of those alerts being triaged by the developers. These four cases are summarized:

- For Linux, 18,016 "parse error" alerts were detected in a report on 2016-11-06 which were all eliminated in the next analysis report conducted on the same day;
- For Firefox, 8,783 "explicit null dereferenced" alerts were detected in a report on 2015-03-05 which were all eliminated together on 2015-04-21;
- For Firefox, 3,085 "misused comma operator" alerts were detected on 2016-08-09 because of a buggy Coverity update [4] which were all eliminated in 2016-08-22;
- For Samba, 2,028 "operands don't affect result" alerts were detected in a report on 2015-08-24 which were all eliminated in the next analysis report conducted on the same day.

We have identified these four cases when manually inspecting the count of newly-detected alerts in each analysis report. The number of newly-detected alerts of a certain alert type is unusually high in these cases, and therefore, is an anomaly. Moreover, the alerts were eliminated together in the same subsequent report which is in the same day of detection in two of the cases. One explanation behind these four cases could be an error in the tool configuration which makes the alerts unactionable. However, given that these cases are the outliers for the respective projects over a long period and the large number of alerts generated in these reports would skew the results of the analysis, we have discarded these alerts from our analysis.

(d) Coverity may add prefixes (e.g. 'home/scan/') with filenames that can vary in different analysis reports depending on how the analysis was performed. We manually identified these prefixes and corrected the filenames to map with their respective locations in the repository. Through these process, we ensure that there is no single alert with multiple file paths (duplicates) in our data.

(e) External File alerts: We have observed that there can be alerts for a project on a file that never existed in the history of the 'master' branch of the project repository. We assume that these files can be compiler include files, third-party library files, or generated files. Developers from Firefox and Kodi have confirmed the correctness of our assumption. As we cannot (or, do not) trace back to commits of a files that is not in the master branch of the project repository, we do not have the data on how and by whom those alerts were handled. Therefore, we discard these alerts from our analysis. Table III gives an overview of discarded external file alerts.

TABLE III
DISCARDED ALERTS THAT AFFECT NON-PROJECT FILES

Project	External File Alerts	Type of files
Linux	21 (0.1%)	compiler include files
Firefox	6,731 (34.2%)	complier include files, binding files, generated codes
Samba	3,953 (48.6%)	library files, other branches
Kodi	567 (19.6%)	build files, compiler include file
Ovirt-engine	103 (3.4%)	generated codes

We **validate 39,495 alerts** in this step that we use in our analysis in the rest of this paper.

C. Step 3: Collect Commit History

For each validated alert in Step 2, we refer to the file where the alert is detected as ***'affected file'***. We collect full commit history of the affected file. For each commit, we also retrieve the date and time when the commit was merged into the master branch (***'merge date'***). Furthermore, we also collect information on all code changes made in a commit (***'commit diff'***). We process the commit diff to investigate how many files were modified in the commit, and the modified codes.

V. DATA ANALYSIS: METRICS & METHODS

In this section, we explain the terminology and repeatable instructions for computing metrics and classifications we use in our analysis:

1) ***Actionable Alert****:* We define actionable alert as an alert that developers fix through code changes. Figure 1 gives an overview of our automated technique for classifying an alert as actionable. The automated technique works as follows: we check if there is any commit on the affected file within the time an alert was first detected and the time the alert was eliminated. If yes, we check the date when the commit(s) was merged into the master branch that Coverity analyzes. We check if the merge date is within the time of last detection and the time of elimination of the alert. If yes, we further check if the commit(s) deletes the file or its code diff contains any alert suppression keyword according to Coverity documentation.

[4]https://bugzilla.mozilla.org/show_bug.cgi?format=default&id=1293956

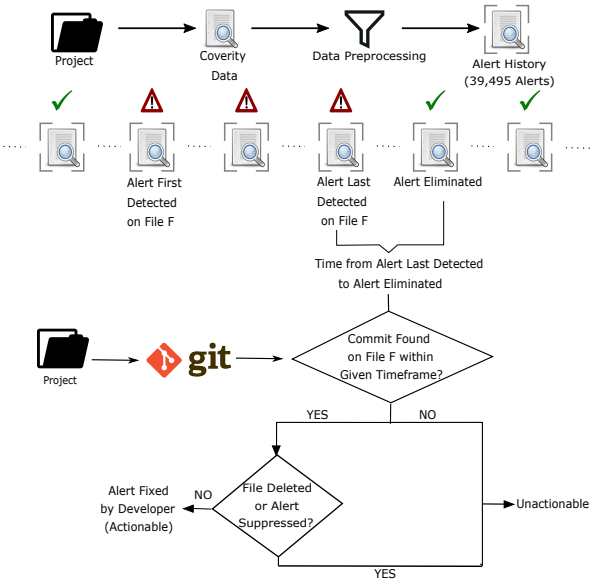

Fig. 1. Automatically identifying actionable alerts through alert detection history and affected file's commit history

If no, we assume the code change made in the commit(s) has contributed in fixing the alert, and therefore, the alert is actionable. In other cases, we classify the alert as unactionable.

Regarding the New alerts (Coverity detected the alert in the last analysis report we collected) that is alive for less than the median lifespan of all actionable alerts for that project, we do not classify them as either actionable or unactionable with the rationale that the time frame for these alerts are not sufficient to make a decision.

*2) **Fix Commit**:* For an actionable alert, if we can identify the specific commit that fixed an alert, we refer to the commit as 'fix commit'. If we find there is only one commit merged into the main code base within the time an alert is last detected and first eliminated, we classify that as a fix commit. If there is more than one commit within that time window, we further look for the keyword 'coverity' or 'CID' [5] in the commit messages, and if there is *only a single commit* with any of the two keywords in the commit message, we classify that commit as a fix commit. Otherwise, we are unable to track fix commits for an alert. Therefore, while measuring fix complexity, we only analyze the alerts that we could track a fix commit.

*3) **Lifespan**:* The lifespan is approximated to be the time of how long an alert remains in the main code base. To measure this, we take the time difference in days between the last and the first time an alert was detected. For the alerts that changed IDs due to file renaming, we calculate the lifespan by taking the difference between the first detection date of the old alert id and the last detection date of the new alert id.

*4) **Actionability Rate**:* We compute the actionability rate by dividing the count of actionable alerts by the total count of actionable and unactionable alerts as classified in V-1.

[5]We look for a regex matching with 'cid' followed by an whitespace or digit to ensure minimum false positives.

*5) **Median Rate for Alert Types:** :* We present an analysis on occurrences, actionability, lifespan, and fix complexity for top 10 alert types in Section VI. For C/C++ alert types, we measure the metrics in two steps: 1) take the median count of a metric for all the alerts from that alert type under each project; and 2) take median count of the median rates from the four projects. We only do the first step for Java alert types as we have only one Java project,

*6) **Complexity of Fix Commit**:* We followed Li et al.'s [35] approach to measure complexity of code changes in fix commits. We measure five metrics which are explained below. While the first three metrics are a direct adoption from Li et al.'s work, we also measure the code changes made only in the affected file in which the alert was detected in the last two metrics based on the assumption that changes in other files may be non-related to the fix of the alert.

a *Affected File Count*: The number of files modified in the commit.
b *Net Lines of Code (LOC) Change*: The number of LOC that were modified across all the files.
c *Net Logical Change*: "Logical change" is a grouping of consecutive lines changed within a commit. For this metric, we count the number of logical changes across all the files.
d *In-File Lines of Code (LOC) Change*: The number of LOC that were modified only in the affected file.
e *In-File Logical Change*: The number of logical changes only in the affected file.

In measuring these metrics, we also follow Li et al.'s [35] approach of preprocessing the commit diffs by removing blank lines and comment lines [1]. However, unlike Li et al., we do not perform any filtering on the files based on their extension to filter out non-source code files as we observed the fix commits affect 1-3 files on average and therefore, the process would result in diminishing returns.

In the case where developers have fixed multiple alerts through a single fix commit, we **normalized** all the metrics for that commit by a division with the number of alerts fixed in order to truly reflect the complexity of code changes required to fix each single alert in a fix commit.

*7) **Statistical Tests**:* We below describe two statistical tests that we use in our analysis.

a Spearman's Rank Correlation test: The test measures the strength and direction of association between two ranked variables. The test generate two values - a) correlation coefficient (r), and b) statistical significance ($p - value$). The value of correlation coefficient will be between -1.00 (perfect negative correlation) and $+1.00$ (perfect positive correlation) and is interpreted as follows [25]: $0.00 \leq r \leq 0.19$ means a very weak correlation; $0.20 \leq r \leq 0.39$ means a weak correlation; $0.40 \leq r \leq 0.69$ means a moderate correlation; $0.70 \leq r \leq 0.89$ means a strong correlation; $0.90 \leq r \leq 1.00$ means a very strong correlation.
b Mann-Whitney U test: A non-parametric test of the null hypothesis that it is equally likely that a randomly selected value from one sample will be less than or greater than a

TABLE IV
ACTIONABLE ALERTS

Prject	Total Alerts	Eliminated Alerts	Actionable Alerts	Triaged Bug
Linux	17133	10336 (60.3%)	6047 (36.7%)	624 (3.6%)
Firefox	12945	9522 (73.6%)	6193 (48.4%)	1062 (8.2%)
Samba	4186	3055 (73.0%)	1148 (27.4%)	102 (2.4%)
Kodi	2325	1538 (66.2%)	1146 (49.5%)	369 (15.9%)
Ovirt-engine	2906	1302 (44.8%)	905 (31.3%)	75 (2.6%)

TABLE V
TOP 10 ALERT TYPE OCCURRENCES FOR C/C++ PROJECTS

Alert Type	Im-pact	Occurr-ence	Action-ability	Lifespan (days)
Resource leak	H	844.0	49.9%	121.5
Unchecked return value	M	469.0	38.7%	109.5
Logically dead code	M	385.0	44.3%	89.5
Explicit null dereferenced	M	304.0	38.4%	83.2
Dereference after null check	M	273.5	47.2%	178.0
Dereference before null check	M	254.0	62.3%	51.0
Various (a type by Coverity)	M	214.5	33.4%	660.5
Dereference null return value	M	212.0	48.1%	65.0
Uninitialized scalar variable	H	170.0	57.0%	24.5
Missing break in switch	M	141.5	41.6%	173.8

randomly selected value from a second sample. The test generates a $p-value$ that is if less than 0.05, we interpret there is a significant difference between the tested samples.

VI. FINDINGS

We present the findings for each of our research questions:

A. *RQ1: (Occurrence) To what extent do Coverity alerts occur in terms of alert type?*

For this question, we investigate the total number of occurrences and the respective frequency of each alert type. The second column of Table IV shows the number of total alerts for each analyzed project. For the C/C++ projects, the alerts fall under 193 distinct types. However, only 67 alert types (34.7%) occur in all four projects. Figure 2 shows the cumulative distribution of occurrence per percentages of alert types, starting from the most frequent alert types for each project. We see that 80% occurrences of alerts comes from roughly 20% of the alert types (13.4% to 22% across projects, median: 19%). Previously, Liu et al. applied FindBugs on Java projects and found that 10% of the alert types causes 80% of all the alerts. For the single Java project in our dataset, Ovirt-engine, 13% of the alert types causes 80% of the alerts. Table V and VI shows the top 10 alerts types, respectively, for C/C++ projects and for the single Java project. We find that 'Resource Leak' and 'Explicit null derefrenced' are the most occurred alert type for C/C++ projects and the Java project, respectively.

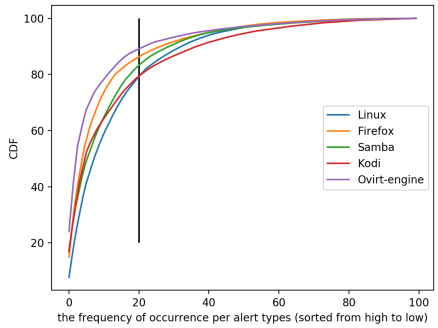

Fig. 2. Cumulative Distribution Functions (CDFs) of the frequency of occurrence for the percentages of alert types for each project. The axes mean X% of alert types cause Y% of alert occurrences.

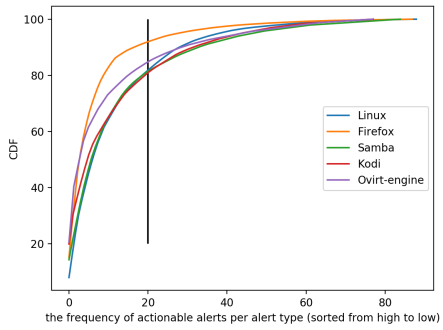

Fig. 3. Cumulative Distribution Functions (CDFs) of the frequency of actionability for the percentages of alert types for each project. The axes mean X% of alert types cause Y% of actionable alerts

TABLE VI
TOP 10 ALERT TYPE OCCURRENCES FOR JAVA PROJECT

Alert Type	Im-pact	Occurr-ence	Action-ability	Lifespan (days)
Explicit null dereferenced	M	700.0	10.9%	136.5
Useless call	M	494.0	1.6%	4.0
Dereference null return value	M	388.0	48.2%	77.0
Bad casts of object references	L	197.0	90.9%	165.0
Problems with implementation of equals()	L	176.0	13.1%	152.0
Dubious method used	L	97.0	73.2%	139.0
Missing call to superclass	M	95.0	31.6%	39.0
Dereference after null check	M	62.0	69.4%	122.0
Inner class could be made static	L	57.0	40.4%	91.0
Resource leak	H	55.0	29.1%	56.5

B. *RQ2: (Actionability) Which Coverity alerts are fixed by the developers through code change?*

We find that the actionability rate for the five projects vary between 27.4% to 49.5% as shown in the fourth column of Table IV. The result is higher than previously reported [21], [34]. Furthermore, for C/C++ projects, we find only 47 alert types (24.4%) to have at least one actionable alert in each of the projects. Figure 3 also indicates that top 20% alert types constitutes 80% actionable alerts similar to what we have seen for occurrences (9.5% to 21% across projects, median: 19.4%). Table V shows that the actionability rate varies between 33.4% to 62.3% for the top C/C++ alert types. Table VI shows the actionability rates for the top Java alert types. We see while

TABLE VII
CATEGORIES OF UNACTIONABLE ALERTS

Project	Total Alerts	Unactionable Alerts	Alive alerts	Triaged False Positive	Triaged Intentional	File Deleted	Suppressed in Code	Eliminated through undetermined ways
Linux	17133	10446 (61.0%)	4530 (26.4%)	738 (4.3%)	781 (4.6%)	342 (2.0%)	152 (0.9%)	3903 (22.8%)
Firefox	12945	6590 (50.9%)	1256 (9.7%)	963 (7.4%)	982 (7.6%)	215 (1.7%)	90 (0.7%)	3084 (23.8%)
Samba	4186	3038 (72.6%)	904 (21.6%)	106 (2.5%)	73 (1.7%)	80 (1.9%)	5 (0.1%)	1870 (44.7%)
Kodi	2325	1168 (50.2%)	373 (16.0%)	84 (3.6%)	319 (13.7%)	95 (4.1%)	0 (0.0%)	297 (12.8%)
Ovirt-engine	2906	1988 (68.4%)	107 (3.7%)	1280 (44.0%)	203 (7.0%)	26 (0.9%)	10 (0.3%)	362 (12.5%)

TABLE VIII
MEDIAN ALERT LIFESPAN (IN DAYS)

Project	Actionable Alerts	Alerts Marked as Bug	Unactionable Eliminated
Linux	245.0	184.0	231.0
Firefox	124.0	64.0	174.0
Samba	39.5	200.0	46.0
Kodi	36.0	2.0	56.0
Ovirt-engine	96.0	43.0	152.0

'Bad casts of object reference' has high actionability (90.9%), 'Useless call' on the other hand have very low actionability (1.6%) among the top Java alert types.

Table VII shows a categorization of unactionable alerts based on how they were eliminated. We find a large number of unactionable alerts got eliminated in a way that we could not determine. The possible ways could be:

- Developers used tuning features (modeling file[6], ignore alerts, or ignore files) to suppress unactionable alerts;
- An update in the Coverity tool; or
- Code change in other areas than the affected file.

C. *RQ3: (Lifespan) How long do Coverity alerts remain in the code before being fixed?*

We find the median lifespan of actionable alerts to vary between projects (36 to 245 days) as shown in Table VIII. We further investigated if the alerts that developers triage as Bug differs in their lifespan from the actionable alerts that are not triaged as Bug as we conjecture that the alerts triaged as Bug are deemed as severe defects by the developers and require mandatory addressing. For Linux, Firefox, and Kodi, we find that alerts triaged as Bug have significantly shorter lifespan ($p < .05$ using Mann-Whitney U test) than other actionable alerts while for Samba the relation is in the opposite direction. For Ovirt-engine, we do not find a significant difference. Table V and VI show the median lifespan for top alert types.

D. *RQ4: (Fix complexity) What is the complexity of the code changes that fix a Coverity alert?*

Table IX shows all the metrics we use to measure fix complexity of the code changes to fix Coverity alerts. All the metrics indicate low complexity of the code changes for Linux, Samba, and Kodi. However, for Firefox and Ovirt-engine, we see that net LOC change and net Logical change is high despite changes in the affected file being still low in complexity. A possible reason could be that for these two projects, developers fixed the alerts while making other changes to the files which is why we find a larger change in totality (i.e. alert fix is not the *only* action in the commit). To control for this, we further investigate only the alerts that are triaged as Bug as we conjecture developers would push a dedicated commit while fixing an alert they identified as Bug. Table IX also shows the metrics to measure fix complexity only for the alerts that were marked as Bug, We see that for all the projects, including Firefox and Ovirt-engine, all the metrics indicate low complexity code changes. Therefore, our results suggest that fixing Coverity alerts require low complexity code change.

As we could not identify the fix commit for all the actionable alerts, our result might get biased due to the dominance of few alert types while measuring fix complexity. Therefore, we further investigate fix complexity across alert types. We could identify at least one fix commit for 173 alert types across five projects. In Table X, we present the findings for top 10 alert types in terms of the number of fix commits that we could track for. We find that each alert type exhibits similar pattern of complexity (1 to 4.5 units of logical changes across all files) as the overall fix complexity of all the alerts.

E. *RQ5: Do Coverity alerts with higher severity have shorter lifespan?*

We correlate impact levels (High, Medium, Low) set by the Coverity tool for each of the alert with the actionable alerts' lifespan. Specifically, we compute Spearman's correlation for each of the five projects where the null hypothesis is that there exists no monotonic correlation between severity and lifespan of an alert. Table XI shows the Spearman's coefficients alongside median lifespan for alerts of each impact levels. We do not find a significant correlation between severity and lifespan for any project, except Kodi. For Kodi, we get $p < .05$ which suggests that the null hypothesis is rejected and there exists a weak negative correlation (as hypothesized) between severity and lifespan which tells us that alerts with higher severity get fixed faster than lower severe ones.

F. *RQ6: Do Coverity alerts with lower fix complexity have shorter lifespan?*

We correlate fix complexity with lifespan for alerts across all five projects using Spearman's correlation where the null hypothesis is that there is no monotonic correlation between an alert's fix complexity and lifespan. However, as we have

[6]Coverity modeling file for project Samba: https://github.com/samba-team/samba/blob/master/coverity/coverity_assert_model.c

TABLE IX
FIX COMPLEXITY

Project	Fix Commit tracked	Affected Files	net LOC Change	net Logical Change	In-File LOC change	In-File Logical change
All Alerts						
Linux	2299	1.0	4.0	1.5	3.0	1.0
Firefox	1835	3.0	40.7	11.0	5.0	1.7
Samba	639	1.0	3.0	1.0	2.0	1.0
Kodi	469	1.0	6.2	2.0	4.0	1.0
Ovirt-engine	666	1.5	24.9	5.5	7.0	2.0
Alerts Marked as Bug						
Linux	294	1.0	2.0	1.0	2.0	1.0
Firefox	345	1.0	9.5	3.0	4.0	1.5
Samba	27	1.0	5.0	1.0	4.0	1.0
Kodi	46	1.5	6.7	3.0	2.0	1.0
Ovirt-engine	68	1.0	4.0	2.0	4.0	1.0

TABLE X
FIX COMPLEXITY FOR TOP ALERT TYPES (IN TERMS OF THE NUMBER OF TRACKED FIX COMMITS)

Alert Type	Fix Commit tracked	Affected Files	net LOC Change	net Logical Change	In-File LOC change	In-File Logical change
Uninitialized scalar field	497	2.5	13.3	4.5	4.0	1.0
Resource leak	493	1.0	6.0	2.0	4.0	2.0
Logically dead code	363	1.0	5.0	1.5	4.0	1.0
Unchecked return value	337	1.0	16.0	3.8	5.0	1.4
Explicit null dereferenced	296	1.1	12.5	3.0	3.3	1.0
Dereference null return value	284	1.0	10.0	2.0	6.0	2.0
Uninitialized scalar variable	245	1.0	2.0	1.0	2.0	1.0
Dereference before null check	227	1.0	6.0	2.0	4.0	2.0
Dereference after null check	210	1.0	6.2	1.5	4.6	1.0
Uninitialized pointer field	159	2.0	13.3	4.1	5.0	1.0

TABLE XI
CORRELATION BETWEEN ALERT'S SEVERITY AND LIFESPAN

Project	r	High Alert Lifespan	Medium Alert Lifespan	Low Alert Lifespan
Linux	-0.02	217.0	248.0	206.0
Firefox	0.0	154.0	89.0	105.5
Samba	-0.01	36.0	56.0	18.0
Kodi*	-0.29	2.0	26.5	416.0
Ovirt-engine	-0.03	89.0	77.0	129.0

multiple metrics to measure fix complexity, we compute correlation with lifespan individually for each of the metrics:

- Affected Files: We get $p < .05$ for each of the projects *except* Kodi and Samba to reject the null hypothesis. Linux, Firefox, and Ovirt-engine has a positive correlation coefficient of 0.05, 0.17, and 0.40 respectively.
- Net LOC change: We get $p < .05$ for each of the projects to reject the null hypothesis. Linux, Firefox, Samba, Kodi, and Ovirt-engine has a positive correlation coefficient of 0.09, 0.20, 0.16, 0.15, and 0.44 respectively.
- Net Logical change: We get $p < .05$ for each of the projects *except* Linux to reject the null hypothesis. Firefox, Samba, Kodi, and Ovirt-engine has a positive correlation coefficient of 0.20, 0.12, 0.10, and 0.44 respectively.
- In-File LOC change: We get $p < .05$ for each project to reject the null hypothesis. Linux, Firefox, Samba, Kodi, and Ovirt-engine has a positive correlation coefficient of 0.06, 0.12, 0.11, 0.22, and 0.18 respectively.
- In-File Logical change: We get $p < .05$ for each project *except* Linux and Samba to reject the null hypothesis. Firefox, Kodi, and Ovirt-engine has a positive correlation coefficient of 0.07, 0.20, and 0.20 respectively.

We find that Firefox and Ovirt-engine show a significant positive correlation between fix complexity and lifespan in all the metrics. However, the strength of the correlations vary between very weak correlation to moderate correlation.

VII. LESSONS LEARNED

We find actionability rate in our case studies to be higher than previously reported in similar studies [21], [36]. One explanation is that Coverity focuses only on security and reliability alerts unlike SonarQube in [21] which also looks for code smells and minor issues. Therefore, Coverity detects a smaller number of alerts with higher probability of actionability. Moreover, developers may leverage the tuning features of Coverity to reduce false positives. Prior work [41] conjectures that SATs need to have an actionability rate around 90% to be trusted by the developers. Sadowski et al. [41] shows Google's in-house static analysis checkers with strict actionability requirement to achieve an actionability rate around 95%. However, in this paper, we analyze projects from different organizations that use a commercial external tool. Given that 1) Coverity is a reputed tool with a narrow focus towards security and reliability errors; and 2) developers from large-scale open source projects are actively monitoring the alerts and can tune the tool configurations as needed — **our empirical evidence shows that the actionability rate**

of SATs in the real word is around 36.7% (median rate across five case studies), better than previously reported, but still lags far behind the ideal.

We also find that fixing Coverity alerts generally require small changes in code. Another explanation could be that developers only tend to fix the easy-to-fix alerts. However, we also find low complexity fixes for the alerts that were triaged as a Bug (we conjecture that developers triage an alert as a Bug based on severity and before any effort estimate of the fix). A complaint against SATs is that alerts can be expensive to fix [40]. Therefore, SATs can motivate the developers in quickly addressing the alerts by showing an estimate of effort for the fix. Prior work has also shown that the tool providing resolution suggestion alongside the alert would be useful for the developers [30], [32]. Liu et al. [36] has proposed an approach to automatically identify fix patterns of SAT alerts through convolutional neural network that leverages past fixes of alerts from the developers. We conjecture that automated fix of SAT alerts is a promising field of future research as our empirical evidence shows that **fixes of static analysis alerts are low in complexity** – 1 to 2 units of logical changes in the affected file.

We find varying results for the lifespan of actionable alerts across projects and how severity and fix complexity correlate with the lifespan. One explanation is that projects have different development structure and different monitoring policy for Coverity alerts. However, as a common pattern, we find four out of five projects do not prioritize alerts based on their severity level set by the tool. Conversely, three out of five projects fix alerts that are triaged as a bug by their developers faster than the rest of the actionable alerts. While identification of actionable alerts from SATs is a well-studied field in the literature (e.g. [17], [18], [29], [39]), **future research should also focus on prioritizing alerts that can be a critical issue, for example security defects, so that developers can know which alerts require immediate addressing.**

VIII. Threats to Validity

Our methodology assumes that if there is any code change in the affected file within the time the alert was last detected and first eliminated, developers have made code changes *to fix* the alert. Otherwise, we classify the alert as unactionable. There are two threats to this methodology: 1) The code change in the affected file can be unrelated to the alert, and 2) Developers can make code changes in files other than the affected one to fix the alert in which case we would be falsely classifying the alert as unactionable. However, as we do not have the data on the associated events and the exact code location of each event for an alert (mentioned in III-B), we cannot further investigate this threat.

Our analysis of fix complexity relies on accurate tracking of fix commits for an alert. To validate fix commit tracking, the first author manually looked at 25 randomly selected alerts from each project for which our automated technique could track a fix commit. The author looked at the alert type and the commit message to determine if the commit *fixes* the alert. From Linux, Firefox, Samba, Kodi, and Ovirt-engine, the author could validate respectively 18, 12, 21, 19, and 21 commits that they were a fix to the alert as explained in the commit message. For the rest of the cases, the commit messages do not specifically address the alert and therefore, we could not validate if they are indeed a fix for the alert.

Furthermore, there can be inaccuracies and inconsistency in the data that we collected from Coverity. We explained our data preprocessing steps in Section IV to minimize this threat. Any inconsistency between the timestamps of an alert's detection history and the affected file's commit history can also harm our analysis.

External Validity: SATs can differ in their underlying technology and objectives. SATs can adopt various *independent* techniques, such as pattern matching, symbolic execution, data flow analysis, and formal verification while the tools can also vary in their degree of sensitivity to control-flow, context, and execution path analysis [22]. Therefore, our choice of Coverity as the targeted SAT poses a threat to how much our findings are representative of static analysis alerts in general. As a commercial tool, Coverity's underlying technology is not public. However, relevant materials suggest that Coverity uses multiple techniques, including, but not limited to: pattern matching, statistical, data-flow, inter and intra-procedural analysis [10], [20]; and covers a broad range of security and reliability errors [10], [22] (confirmed by our study). Furthermore, Coverity encompasses FindBugs [12] rules for Java where FindBugs is a well-known open source SAT. Coverity is regarded as state-of-the-art [19], [22] and is used widely in the industry [23]. Therefore, we believe the threat is minimal and our findings represent how developers in the real-world are acting on SAT alerts in general.

IX. Conclusion

In this paper, we empirically study five open source projects as case studies that have been actively using Coverity, a static analysis tool, with an aim to understand how developers act on static analysis alerts. We find that the portion of total alerts that developers fix through code changes vary between 27.4% to 49.5% across projects. We also find that the developers generally take a long time to fix the alerts despite the fixes being low in complexity, and the factors that affect the time may vary from project to project. Our findings provide the latest empirical evidence on the actionability of static analysis alerts. Furthermore, based on our findings, we suggest that - 1) future research should focus on prioritizing critical alerts that require immediate addressing of the developers; and 2) toolmakers may consider motivating the developers in fixing the alerts by providing an estimate of required effort and resolution suggestions.

X. Acknowledgements

We thank the reviewers and the RealSearch group for their feedback. This material is based in part upon work supported by the National Science Foundation under grant number 1451172.

REFERENCES

[1] C++, C, C#, Java and JavaScript code guidelines. https://named-data.net/codebase/platform/documentation/ndn-platform-development-guidelines/cpp-code-guidelines/.

[2] Clang-static-analyzer. https://clang-analyzer.llvm.org/.

[3] Coverity scan. https://scan.coverity.com/.

[4] Findbugs. http://findbugs.sourceforge.net/.

[5] Git–distributed-is-the-new-centralized. https://git-scm.com/.

[6] Semmle. www.semmle.com.

[7] Sonarqube. https://www.sonarqube.org/.

[8] Travis ci. https://travis-ci.com/.

[9] Veracode. www.veracode.com.

[10] Ali Almossawi, Kelvin Lim, and Tanmay Sinha. Analysis tool evaluation: Coverity prevent. *Pittsburgh, PA: Carnegie Mellon University*, pages 7–11, 2006.

[11] Parasoft Arthur Hicken, Cheif Evangelist. What is shift-left testing? https://blog.parasoft.com/what-is-the-shift-left-approach-to-software-testing.

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

[13] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

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

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

[16] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[17] Foteini Cheirdari and George Karabatis. Analyzing false positive source code vulnerabilities using static analysis tools. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 4782–4788. IEEE, 2018.

[18] Bharti Chimdyalwar, Priyanka Darke, Anooj Chavda, Sagar Vaghani, and Avriti Chauhan. Eliminating static analysis false positives using loop abstraction and bounded model checking. In *International Symposium on Formal Methods*, pages 573–576. Springer, 2015.

[19] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 332–343. IEEE, 2016.

[20] Dr. Jared DeMott. Static code analysis: Scan all your code for bugs. https://www.youtube.com/watch?v=Heor8BVa4A0&t=641s.

[21] Eduardo Monteiro Edna Canedo Welder Luz Gustavo Pinto Diego Marcilio, Rodrigo Bonifcio. Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In *the 27th IEEE/ACM International Conference on Program Comprehension (ICPC2019)*, 2019.

[22] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[23] Enlyft. Companies using coverity. https://enlyft.com/tech/products/coverity.

[24] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. Cloudbuild: Microsoft's distributed and caching build service. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 11–20. ACM, 2016.

[25] Jim Fowler, Lou Cohen, and Phil Jarvis. *Practical statistics for field biology*. John Wiley & Sons, 2013.

[26] Philip J Guo and Dawson R Engler. Linux kernel developer responses to static analysis bug reports. In *USENIX Annual Technical Conference*, pages 285–292, 2009.

[27] Sarah Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 41–50. ACM, 2008.

[28] Sarah Heckman and Laurie Williams. A model building process for identifying actionable static analysis alerts. In *2009 International Conference on Software Testing Verification and Validation*, pages 161–170. IEEE, 2009.

[29] Sarah Heckman and Laurie Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4):363–387, 2011.

[30] Nasif Imtiaz, Akond Rahman, Effat Farhana, and Laurie Williams. Challenges with responding to static analysis tool alerts. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 245–249. IEEE Press, 2019.

[31] Nasif Imtiaz and Laurie Williams. 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*, page 12. ACM, 2019.

[32] Brittany Johnson, Rahul Pandita, Justin Smith, Denae Ford, Sarah Elder, Emerson Murphy-Hill, Sarah Heckman, and Caitlin Sadowski. A cross-tool communication study on program analysis tool notifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 73–84. ACM, 2016.

[33] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.

[34] Sunghun Kim and Michael D Ernst. Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54. ACM, 2007.

[35] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215. ACM, 2017.

[36] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, 2018.

[37] Synopsys Mel Llaguno, Open Source Solution Manager. 2017 coverity scan report. open source softwarethe road ahead. https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/SCAN-Report-2017.pdf, lastchecked=26.04.2019.

[38] Hacker News. Twitter outage report. https://news.ycombinator.com/item?id=8810157, 2016.

[39] Hendrik Post, Carsten Sinz, Alexander Kaiser, and Thomas Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 188–197. IEEE, 2008.

[40] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. 2018.

[41] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 598–608. IEEE Press, 2015.

[42] Synopsys. Coverity. https://en.wikipedia.org/wiki/Coverity.

[43] Synposis. How to write a function model to eliminate a false positive in a c application. https://community.synopsys.com/s/article/How-to-write-a-function-model-to\protect\discretionary{\char\hyphenchar\font}{}{}eliminate-a-false-positive-in-a-C-application.

[44] Synopsys Editorial Team. Coverity report on the 'goto fail' bug. https://www.synopsys.com/blogs/software-security/apple-security-55471-aka-goto-fail/, February 2014.

[45] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 334–344. IEEE, 2017.