The Journal of Systems and Software 158 (2019) 110427

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

An empirical study of security warnings from static application security testing tools

Bushra Aloraini[a,b,*], Meiyappan Nagappan[a], Daniel M. German[c], Shinpei Hayashi[d], Yoshiki Higo[e]

[a] David R. Cheriton School of Computer Science, University of Waterloo, Canada
[b] College of Computer and Information Sciences, Princess Nora bint Abdul Rahman University, Saudi Arabia
[c] School of Engineering, University of Victoria, Canada
[d] School of Computing, Tokyo Institute of Technology, Japan
[e] Graduate School of Information Science and Technology, Osaka University, Japan

ARTICLE INFO

Article history:
Received 16 May 2019
Revised 21 September 2019
Accepted 23 September 2019
Available online 23 September 2019

Keywords:
False positives
Security warnings
Software vulnerability
Static application security testing tools

ABSTRACT

The Open Web Application Security Project (OWASP) defines Static Application Security Testing (SAST) tools as those that can help find security vulnerabilities in the source code or compiled code of software. Such tools detect and classify the vulnerability warnings into one of many types (e.g., input validation and representation). It is well known that these tools produce high numbers of false positive warnings. However, what is not known is if specific types of warnings have a higher predisposition to be false positives or not. Therefore, our goal is to investigate the different types of SAST-produced warnings and their evolution over time to determine if one type of warning is more likely to have false positives than others. To achieve our goal, we carry out a large empirical study where we examine 116 large and popular C++ projects using six different state-of-the-art open source and commercial SAST tools that detect security vulnerabilities. In order to track a piece of code that has been tagged with a warning, we use a new state of the art framework called *cregit*$^{+}$ that traces source code lines across different commits. The results demonstrate the potential of using SAST tools as an assessment tool to measure the quality of a product and the possible risks without manually reviewing the warnings. In addition, this work shows that pattern-matching static analysis technique is a very powerful method when combined with other advanced analysis methods.

1. Introduction

Software bugs significantly reduce system quality and greatly increases the cost. It is estimated that software bugs cost the worldwide economy US$1.1 trillion in 2016 (Baptista, 2017). Some of the bugs are security vulnerabilities, which are exploitable code errors that may be manipulated to compromise the intended operation of the software in a malicious way.

Exploiting vulnerabilities may lead to catastrophic consequences resulting in harm to organizational assets, financial loss, or harm to individuals (Common Weakness Enumeration). Static Application Security Testing (SAST) tools can potentially detect software vulnerabilities early during the coding phase. They do this by statically examining the code, and providing the developer with warnings in the code that could potentially be vulnerabilities. The SAST tools classify each warning into one of many types and assign severity to them. Using such tools could lead to cost savings (Soni, 2006). However, SAST tools have not been well adopted, as they may produce a high rate of false positive alarms since most warnings do not indicate real vulnerabilities (Christakis and Bird, 2016; Johnson et al., 2013).

Hence, an understanding of the warnings that are generated by SAST tools, their characteristics, including their distribution across various types, and decay time, could be useful in improving the quality of the SAST tools. For instance, if we find that many warnings of a specific type are indeed false positives, then the developers of SAST tools could then build a better detection algorithm for that type of warning or reduce its severity score. Alternatively, based on our approach and potentially our results, developers could ignore the warnings that are more likely to be false positives, more readily.

* Corresponding author.
E-mail addresses: baloraini@uwaterloo.ca, bsaloraini@pnu.edu.sa (B. Aloraini), mei.nagappan@uwaterloo.ca (M. Nagappan), dmg@uvic.ca (D.M. German), hayashi@c.titech.ac.jp (S. Hayashi), higo@ist.osaka-u.ac.jp (Y. Higo).

https://doi.org/10.1016/j.jss.2019.110427
0164-1212/© 2019 Published by Elsevier Inc.

Since understanding the characteristics of warnings from SAST tools is an important problem, we are not the first to examine them. Some studies have been conducted to evaluate warnings that are generated by SAST tools, yet they mostly rely on test cases. For instance, Díaz and Bermejo (2013) have provided an evaluation of nine SAST tools against SAMATE Reference Dataset test suites to analyze the performance of SAST tools in detecting security vulnerabilities. However, software projects in this dataset and their vulnerabilities are well documented and freely available to everyone. Hence, evaluating the goodness of a SAST tool on such datasets have limitations since, the tools could have been trained to detect this exact vulnerability in the dataset. Testing the tools and examining the characteristics of the warnings produced on software projects in the wild where one does not know where vulnerabilities exist, can be more useful. In fact, a study by the National Security Agency (NSA) (National Security Agency Center for Assured Software, 2011) has emphasized the importance of evaluating SAST tools against real software projects to precisely predict the frequencies of warnings and detected vulnerabilities outside of controlled studies.

One such study that examined real-world software projects that are not part of controlled studies is by Di Penta et al. (2009). In their study, they examined three networking systems to study warnings detected by three freely available SAST tools. They used the *ldiff* tool (Canfora et al., 2009), to trace the source code lines across subsequent commits, in order to see if a warning in the source code in one commit is removed in a subsequent commit. Their study obtained more realistic results since it evaluated such tools against real-world projects. For this reason, we extend the study by Di Penta et al. (2009) and investigate the nature of software warnings as generated by SAST tools for C++ projects. Instead of studying only three networking systems, we apply the study on 116 real projects that belong to different categories, as well as, four free and open source SAST tools and two commercial closed source SAST tools. We use *cregit*$^+$, which is a more recent version of the state-of-the-art tool called cregit (German et al., 2019). *cregit*$^+$ takes full advantage of version control, that helps to track file renames. Also, it tracks the source code at a finer granularity (at the token level rather than line level), which helps to detect line split and merge. Our working assumption (like that of Di Penta et al., 2009) is that if a warning was not removed and remains in the same piece of code across many versions for a long time (in our case more than five years), then the warning of that type may not be that important, since the warning was not considered worth removing under any circumstance. Thus, that warning is more likely to be a false positive.

More specifically, the contributions of our study are as follows:

- We first examine the distribution of the various types of SAST-produced warnings through time.
- We investigate how many of the warnings of each type remain over time.
- We also examine the rate of decay for warnings belonging to different categories and produced by various SAST tools. This is the time interval between the discovery and removal of a warnings, to see how long different types of warnings tend to stay in the project.
- Finally, we investigate the distribution of real-world vulnerability categories that are fixed by the project team in the studied projects to see if the results are consistent with the rest of the study.

2. Related work

Multiple empirical studies have been performed to understand SAST performance (Díaz and Bermejo, 2013; Kratkiewicz and Lippmann, 2005; Torri et al., 2010; Zitser et al., 2004); however, many of the related work used benchmarks or test cases (some with small-sized systems). As mentioned in Section 1, test cases are software projects with known vulnerabilities that have been curated and made available to everyone. Hence, the SAST tools could be tuned to detect the vulnerabilities in these test cases. For instance, Zitser et al. (2004) have conducted a study on three open source programs; BIND, WU-FTPD, and Sendmail that contain 14 exploitable buffer overflow vulnerabilities to test five open source SAST tools. While Torri et al. (2010) provided a similar study, it was generalized for multiple vulnerability types that are present in five embedded systems. Kratkiewicz and Lippmann (2005) have evaluated five SAST tools using a corpus of 291 small C program test cases. Díaz and Bermejo (2013) run nine SAST tools against two of SAMATE reference dataset test suites for C language. All of the research mentioned above was conducted using test cases having real-world vulnerabilities that are known in advance, in order to assess the false positives. The main idea of our work, however, is to study the historical evolution captured in the repositories that are not part of any standard dataset to evaluate the false positives among the SAST-produced warnings on software projects in the wild.

Few papers involve manual examination to evaluate SAST-produced warnings (Ayewah et al., 2007). Ayewah et al. (2007) run FindBugs against multiple large projects, such as the JDK, Glassfish J2EE server from Sun, and some portions of Google's Java codebase. The authors found that Findbugs reports real but trivial bugs. They adopted manual analysis to evaluate bug warnings produced by FindBugs, while we use a tool for tracing lines of code to extract the extent of false positives, which makes our analysis scalable to trace warnings in hundreds of software projects. Also, we try to understand how false positives from different SAST tools may relate to different types of warnings using a well-known security vulnerability taxonomy (Tsipenyuk et al., 2005).

Edwards and Chen (2012) examined historical releases of four large-scale projects; Sendmail, Postfix, the Apache httpd, and OpenSSL. The main goal of the study is to investigate warnings generated by three SAST tools; the HP Fortify Source Code Analyzer (SCA), IBM Rational AppScan, and Klocwork Insight, and vulnerabilities published in the common vulnerabilities and exposures (CVE) dictionary. The authors examined the discovery rate of the vulnerabilities appearing in the CVE database for the four projects and they correlated that to the output of SAST tools. The authors observed that the number of vulnerabilities does not always decrease with each new release. They also observed that the rate of discovery of vulnerabilities begins to drop three to five years after the initial release. The study showed that the change in number and density of warnings generated by SAST tools is indicative of the change in the rate of discovery of vulnerabilities for new releases. Also, the authors demonstrated that SAST tools can be used to make some assessments of risk even without a human review of the warnings. However, they considered all the warnings in aggregate and not tracing each warning to see if the warning ends up being a bug or even if it is removed, while our research traces SAST-produced warnings across historical versions of projects. Additionally, the goal of their project is to see if the number of warnings correlates with actual vulnerabilities, and not about studying the characteristics of each warning type.

Another research direction has focused on tracing SAST-produced warnings over time (Di Penta et al., 2009; Kim and Ernst, 2007; Spacco et al., 2006). Spacco et al. (2006) proposed a technique for tracking bugs across versions by using repository history and Findbugs. Kim and Ernst (2007) introduced a bug warning prioritization method based on the software change history. They used bug warnings generated by three SAST tools for Java: FindBugs, Jlint, and PMD for three programs: Columba, Lucene, and

Scarab. In that study, only bug warnings that are removed by fix-changes are considered to indicate real bugs. They found that bugs that were fixed in fix-changes represent a very small percentage of bug removals, and about 90% of the bug warnings either remain in the program or are fixed in non-fix changes that were considered to be false positive warnings (Kim and Ernst, 2007). We follow a similar approach to understand the evolution of the SAST-produced warnings using historical information. However, Kim and Ernst neglected bug warnings that appear in non-fix changes, we consider all bug warnings in fix or non-fix changes. This is because researchers have emphasized that linking bugs to the comments in the commit logs may produce biased results (Bachmann et al., 2010; Kalliamvakou et al., 2014). For instance, Bachmann et al. (2010) found that the bugs issued in a bug tracking system could be biased since not all bugs are reported to these systems. Another aspect is that sometimes it is difficult to isolate code changes due to bugs from other code changes due to different reasons. For instance, Murphy-Hill et al. (2015) reported that 75% of surveyed developers had to remove features from software to fix bugs, and are therefore not tagged as bugs. Hence, we consider all commits and see if the warnings remain or disappear in subsequent versions.

Di Penta et al. (2009) performed an empirical study to examine the evolution and decay of bugs in three networking systems detected by three freely available SAST tools. The main focus of that study is to investigate warning categories and decay time. The researchers used the *ldiff* framework, tracing the source code changes across subsequent commits to extract needed information. Our research extends their work (Di Penta et al., 2009), and we use a similar method by employing a novel tool to trace the line codes over time. However, our approach is different from Di Penta et al. (2009) approach in multiple aspects:

– We use a different tracing tool, namely *cregit*$^+$ that traces tokens across commits and allowed to detect line split and merge in subsequent versions and keeps track of the refactored files.
– We conduct our study on 116 open source projects that belong to various categories instead of only three networking systems.
– We use free open source and commercial tools to conduct our analysis.
– We adopt a well-known security bug classification to aggregate the results across tools and provide universal and comprehensive results.

3. Definitions

Below, we define some terms that we will use in the rest of the paper and that are key to understanding our analysis. Note that our analysis examines the first available versions of the software projects in 2012 and 2017. We then see if the warnings that existed in 2012 remain in 2017 or not. Note that we do not know if the developers of the software used any of the SAST tools in this study or not. We are not claiming that. We want to see if warnings that were present in the 2012 version of the software were removed for any reason by the developers of the software in a five year time period or not. This is similar to the context under which Di Penta et al., carried out their study (Di Penta et al., 2009).

Below we present the various scenarios of what could happen to warnings from the 2012 version.

– Remained Warnings - When a line of code in the 2012 version also exists in the 2017 version (even if it has moved location within the file, or the line of code was modified (e.g., a variable name was changed)), and in both versions, the same warning exists. These warnings are likely to be false positives (FP), even though the tool identified a potential problem within the line, the warning has not been removed from the line, indicating that no problem has been found so far in it. Note that the typical time to fix a bug is around three years (Canfora et al., 2009; Di Penta et al., 2009).
– Modified Warnings - When a line of code in the 2012 version also exists in the 2017 version, and the warning in the 2017 version is different from the warning in the 2012 version. These warnings are likely to be true positives (TP), as the project developers have made an effort to change the code that led to the disappearance of the original bug warning, yet they have introduced a new bug which indicates incorrect fixes or buggy patches (Yin et al., 2011). For instance, a bug warning might be raised because of the usage of strcpy function call. The strcpy function call copies a source buffer to a destination buffer and it has two arguments destination and source. So, if the source buffer is larger than the destination buffer that might cause a buffer overflow. A developer might change the code by replacing the strcpy function call by strncpy function call that has three arguments (destination, source, and the number of copied bytes from source to destination). However, strncpy function call has other problems. For instance, it doesn't supply null termination at the destination buffer. (This category shows two aspects: on the one hand: the tool was able to spot a real bug, on the other hand: the developer incorrectly fixed that warning)
– Disappeared Warnings - When a line of code in the 2012 version also exists in the 2017 version, but the warning present in the 2012 version disappears and there is no new warning in the 2017 version. We consider that a Disappeared Warning indicates that the line was improved to remove the warning (and potentially do other work), and thus is a true positive (TP).
– Removed Warnings - When a line of code in the 2012 version *does not* exist in the 2017 version and hence the warning from 2012 also does not exist in the 2017 version. We cautiously decided to make no assertion about Removed Warnings (the line no longer exists). The reason is that we do not know if the line was removed due to a bug fix or because the feature is no longer necessary.

In Di Penta et al.'s work (Di Penta et al., 2009), they referred to warnings as 'Vulnerability'. We are not sure if they include *Modified Warnings* in Disappeared Vulnerabilities. In our paper, we chose not to present or discuss results of removed warnings because we do not know why the line was removed. However, the complete dataset from our study that we share (Aloraini et al., 2019) does have the removed warnings classified, if the reader wants to examine them.

4. Research questions

– **RQ1: How are the warnings from each SAST tool distributed across different warning types in 2012 and 2017?** This research question discusses the distribution of warnings detected by various SAST tools across the various warning types at different time periods: 2012 and 2017. This helps us understand the emergence of warnings to see whether the density of a particular warning type shows any positive or negative trend between 2012 and 2017.
– **RQ2: How are true positive and false positive warnings distributed across different warning types and across different SAST tools?** Our goal is to understand the distribution of true positive (TP) and false positive (FP) warnings that belong to different types as reported by SAST tools and have been removed later from the source code. In particular, we are interested in warnings that have been reported by SAST tools in 2012 and their status in 2017. As per our definitions in Section 3, we present the results of *Remained Warnings* (which are likely to

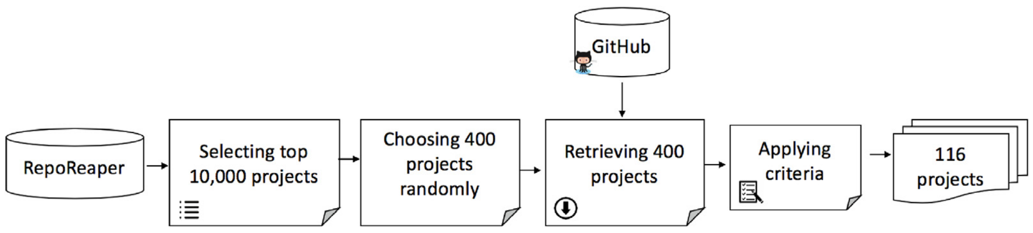


Fig. 1. Case study project selection.

be FP), *Modified Warnings* and *Disappeared Warnings* (which are likely to be TP). Hence, we trace the lines of code across each project's history using *cregit*$^+$.

– **RQ3: How do different types of warnings evolve over time?** We are interested in warning decay, indicating the time interval between detection until removal. To clarify this, we provide this simple scenario: assume that a new project manager decides to run SAST tools in a particular period of time (in our case in 2012). We measure how long it took for the warning to disappear. By knowing this, we would be able to see if warnings of a particular type are removed sooner than others. In RQ2, we examine if the warning disappears by 2017, and now when between 2012 and 2017 it disappears. Hence, we focus on TP warnings (*Modified Warnings* and *Disappeared Warnings*) that were detected by SAST tools and likely fixed in the code. We only consider lines of code that include warnings and have been altered, as this gives us the ability to trace the changes in the lines of code. Whenever we find a warning that belong to an altered line of code and disappeared in the recent version, we trace the line of code that contains the bug warning backward during the history to find the fix time and flag that to be a *bug fix* version.

5. Experimental design

5.1. Selecting case study projects

When we gathered projects to conduct the study, we considered the following four criteria:

- **Real-world projects:** We aim at understanding SAST-produced warnings against real-world large projects. This allows for assessing software project warnings at scale and evaluate SAST tools performance.
- **Programming language:** We choose to analyze C++ projects, since it is ranked as the third most popular programming language based on the TIOBE Index as of May 2019 (TIOBE index for May 2019, 0000). Additionally, C++ is generally more bug prone as well (Ray et al., 2014).
- **Active:** We mine projects that have adequately long and rich historical data, as those projects are more likely to be more mature and active. Thus, we focus on projects with high popularity, as well as projects that have long and continuous development histories for at least five continuous years from 2012 to 2017.
- **Automation:** As we need to run SAST tools on the projects, and some of the SAST tools need to run on top of a build system, we limit retrieved projects to those that use automated build systems. We choose to use projects that adopt the *cmake* build systems.

As shown in Fig. 1, we use projects hosted on GitHub because we want to analyze *real-world projects*. To compose the list of repositories, we make use of the metadata provided by the RepoReaper dataset (Nagappan et al., 2016), since it eliminates the noise (e.g., repositories that are not engineered software projects). Since we want to focus on the *C++ programming language*, we filter the RepoReapers dataset to obtain only projects that were written in C++. We sort the RepoReapers project names by the highest number of stars, then we select the top 15,000 projects on the list. After that, we randomly choose and clone 400 projects from the 15,000 list. To ensure the *activeness* of the projects, we exclude those with fewer than 120 commits.

In addition, we exclude projects that do not have an active development history for at least five continuous years from 2012 till 2017. To fulfill the *automation* criterion, we eliminate all projects that do not adopt the *cmake* build system and could not be built smoothly to run all SAST tools (e.g., due to dependency issues). Finally, we have 116 projects that fulfill all of the above criteria.

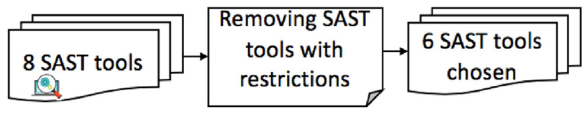


Fig. 2. Selecting static application security testing tools.

5.2. Selecting SAST tools

When selecting the SAST tools that we want to study in this paper (as shown in Fig. 2), we consider their diversity in terms of availability and underlying inference algorithms to provide various types of security warnings. Consequently, we gather the following eight SAST tools that analyze C++ source code: Parasoft C/C++ test (Parasoft C/C++ test), PVS-Studio (PVS-Studio Analyzer), Clang Static Analyzer (Clang Static Analyzer), Cppcheck ((Marjamki)), Flawfinder (Wheeler), RATS (Rough Auditing Tool for Security), Polyspace Bug Finder (Mathworks), and Coverity Static Analysis (Coverity Static Analysis) (academic version). However, we find that some of the SAST tools have restrictions on the number of analyzed lines of code (LOC), such as the Coverity Static Analysis tool (academic version). Also, some of the SAST tools do not generate warnings in an efficient format to analyze. For instance, the Polyspace Bug Finder only shows the warnings using a graphical user interface, which is highly ineffective in analyzing thousands of warnings. Hence, we exclude them and study only six SAST tools. Table 1 summarizes all SAST tools used and related information. Table 1 illustrates different SAST tools with their underlying inference algorithms to infer the presence of security bugs. The inference algorithms that are adopted in the SAST tools are discussed below:

1. **Pattern-matching method** is one of the earliest methods that is used to detect security vulnerabilities (Viega et al., 2002). This method is based on lexical analysis that scans the source code tokens. The goal is to identify potentially dangerous code patterns, such as calls to vulnerable library functions (e.g., strcpy) and some bad practices (e.g., fixed-length array) that may cause bugs. RATS and Flawfinder use this technique solely,

Table 1
Studied static application security testing tools for C++.

Tool name	Version	Availability	Inference algorithm
RATS	2.4	Open source	Pattern-matching
Flawfinder	1.31	Open source	Pattern-matching
Cppcheck	1.76.1	Open source	Value range analysis & data-flow analysis
PVS-Studio	6.13	Commercial*	Pattern-matching & symbolic execution & data-flow analysis annotation-based (automatic)
Parasoft C/C++ test	9.6.1	Commercial	Pattern-matching & abstract interpretation & data-flow analysis
Clang Static Analyzer	3.8	Open source	Symbolic execution & annotation-based & data-flow analysis

* PVS-Studio provides a free academic license.

while other tools, such as Parasoft C/C++ test and PVS-Studio combine pattern-matching with other techniques.

2. **Data-flow analysis method** was developed originally for optimization (Kildall, 1973). Data-flow analysis method collects information about the possible set of values propagating over different program paths. In SAST tools, the information collected by data-flow analysis is used to validate safety properties. For instance, constant propagation is a form of data-flow analysis method. Constant propagation tracks constant values of each program variable. Once a variable, that was folded to a constant value, is used incorrectly in a sensitive operation (e.g., arguments of vulnerable library calls) the method generates a warning. Cppcheck, Clang Static Analyzer, PVS-Studio, and Parasoft C/C++ test adopt this method.
3. **Value range analysis method** has been used in the literature to detect security vulnerabilities (Wagner et al., 2000). Value range analysis uses data-flow analysis to keep track of possible values that variables can be assigned at each point of program execution. Then it solves safety constraints based on the given range of the variables. For instance, if a variable x has range of [3,10] and a variable y has a range of [6,9] throughout the execution of a program. The method infers that the range of the expression $x + y$ is [9,19]. When such expression is evaluated to have an upper bound that exceeds MAXINT (i.e., the maximum value of an integer), this would be flagged as integer overflow warning. Cppcheck performs this type of analysis.
4. **Symbolic execution method** is another method to statically detects security bugs (Xie et al., 2003). The symbolic execution method uses symbolic values when concrete values are not present. Hence, the output values calculated by the method are represented as a function of the input symbolic values. The method binds safety constraints to the symbolic expressions. Then it solves safety constraints that when violated would imply vulnerabilities. For instance, the method may evaluate whether a buffer access is safe by producing safety constraints at every buffer access. Hence, a constraint solver is run to evaluate the values against the constraints (e.g., symbolic constraints between variables like $x < y$). Clang Static Analyzer and PVS-Studio perform this type of analysis.
5. **Abstract interpretation method** is a general method for approximating the behavior of programs (Cousot and Cousot, 1977). The method provides a mathematical guarantee that all properties calculated by the method hold for all possible execution paths of the program. In this method, the concrete values are replaced with abstract ones. For instance, sign analysis can be performed using the Abstract Interpretation method by mapping each variable to a sign instead of mapping it into an integer. Then the instructions can be interpreted by using rules of signs. Parasoft C/C++ test adopts abstract interpretation to detect security bugs.
6. **Annotation-based method** requires program developers to annotate the program code with safety properties in terms of preconditions and postconditions (Evans and Larochelle, 2002). After the code is annotated, the method analyzes the program code starting from the annotated preconditions and validates that the code implementation satisfies the postconditions. For example, developers may annotate a parameter declaration using notnull annotation, which assumes that the passed value for such parameter is not NULL, then the method can validate that. PVS-Studio adopts this technique, and the tool has integrated annotated C/C++ standard functions among others, while Clang Static Analyzer supports this technique, but it requires manual annotation by developers.

5.3. Extracting security warnings

After we choose the SAST tools and projects, we need to extract the warnings that each SAST tool finds in each of the projects. The approach we use for this step is shown in Fig. 3. One of our goals is to see how the warnings evolve over time. Therefore, for each project, we check out two releases of their source code: the first release from 2012 and the first release from 2017. We choose 2012 and 2017 so that we can be fairly certain that warnings from 2012 that are not removed in 2017 do not affect the reliability of the product. We then run each of the SAST tools on 2012 and 2017 releases of every one of the 116 projects and collect the warnings.

We use all available checkers and analysis modules in SAST tools (e.g., when using the Clang Static Analyzer, we enable all available checkers in the tool). Next, we revise all the generated reports to ensure that they are relevant to the actual source code, as we are only interested in source code files that represent the program code, and not source code files for test purposes. Therefore, we exclude all warnings in test files. We infer that from the file path, which usually includes the keyword “test”.

In addition, we only consider warnings that happened in C and C++ source and header files for our analysis. As a result, we find 291,794 warnings in 2012 and 348,441 warnings in 2017 when we consider all the core projects and the linked sub-modules in our dataset, for all the six SAST tools. Finally, since each SAST tool has its own warning format, we produced a consistent and unified format among SAST tools. Therefore, we save every warning from all SAST tools in one format in one file to process later.

5.4. Classifying security warnings

One of the challenges that we face is that the security warning classifications used by the SAST tools are different among the various SAST tools, making it difficult to analyze them in aggregate. Hence, we need to have one consistent classification across the SAST tools to simplify the analysis. We map the generated warnings by different SAST tools to the Seven Pernicious Kingdoms (SPK) classification (Tsipenyuk et al., 2005). The SPK classification provides a taxonomy of common types of security coding errors that might lead to vulnerabilities. This taxonomy is mainly based on the cause of a security vulnerability, but not necessarily the effect, and it focuses on implementation issues.

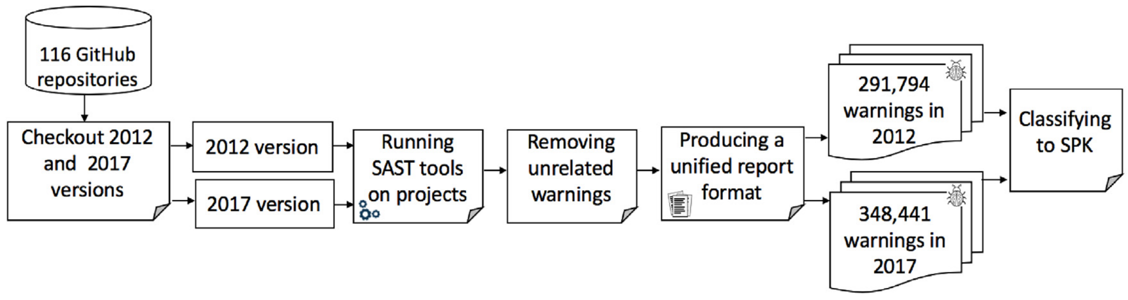

116 GitHub repositories
Checkout 2012 and 2017 versions
2012 version
2017 version
Running SAST tools on projects
Removing unrelated warnings
Producing a unified report format
291,794 warnings in 2012
348,441 warnings in 2017
Classifying to SPK

Fig. 3. Extracting security warnings.

Table 2
Seven pernicious kingdom taxonomy of security warnings (only types related to C++ and those that are language independent are included here).

#	Type	Subtype	#	Type	Subtype
1	Input Validation and Representation **(IVR)**	-Buffer Overflow -Command Injection -Cross-Site Scripting -Format String -HTTP Response Splitting -Illegal Pointer Value -Integer Overflow -Log Forging -Path Manipulation -Process Control -Resource Injection -Setting Manipulation -SQL Injection -String Termination Error Handling -Unsafe JNI -XML Validation	2	Improper Fulfillment of API Contract, or API Abuse **(API)**	- Dangerous Function - Directory Restriction - Heap Inspection - Often Misused: Authentication - Often Misused: Exception Handling - Often Misused: File System - Often Misused: Privilege Management - Often Misused: Strings - Unchecked Return Value
3	Security Features **(SF)**	- Insecure Randomness - Least Privilege Violation - Missing Access Control - Password Management - Password Management: Empty Password in Config File - Password Management: Hard-Coded Password - Password Management: Password in Config File - Password Management: Weak Cryptography - Privacy Violation	4	Time and State **(TS)**	- Deadlock - Failure to Begin a New Session upon Authentication - File Access Race Condition: TOCTOU - Insecure Temporary File - Signal Handling Race Conditions
5	Error Handling **(ERR)**	- Empty Catch Block, Language-Independent - Overly-Broad Catch Block - Overly-Broad Throws Declaration	6	Indicator of Poor Code Quality **(CQ)**	- Double Free - Inconsistent Implementations - Memory Leak - Null Dereference - Obsolete - Undefined Behavior - Uninitialized Variable - Unreleased Resource - Use After Free
7	Insufficient Encapsulation **(ENC)**	- Data Leaking Between Users - Leftover Debug Code - Mobile Code: Non-Final Public Field - Private Array-Typed Field Returned From a Public Method - Public Data Assigned to Private Array-Typed Field - System Information Leak - Trust Boundary Violation	*	Environment **(ENV)**	- Insecure Compiler Optimization - Insecure Configuration Management

Each SAST tool has a list of warning types that they can detect in the documentation, along with a description for each of them. The first author of the paper carefully analyzed these lists and descriptions of warning types and manually classified each warning type into one of the seven types described by the SPK classification (see Table 2 for more detail). For each of the six SAST tools, we determined which tool can identify which type of vulnerability from the SPK classification. The results of that is presented in Table 3.

Table 3
Seven pernicious kingdom categories detected by each tool.

Tool	IVR	API	SF	TS	ERR	CQ	ENC	ENV
RATS	√	√	√	√	✗	√	✗	✗
Flawfinder	√	√	√	√	✗	√	✗	✗
Cppcheck	√	√	✗	√	√	√	✗	✗
PVS-Studio	√	√	✗	√	√	√	√	√
Parasoft C/C++ test	√	√	√	√	√	√	√	√
Clang Static Analyzer	√	√	√	√	✗	√	✗	✗

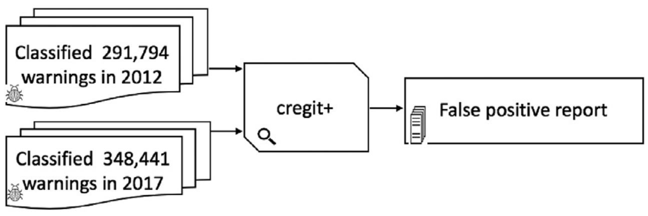

Classified 291,794 warnings in 2012
Classified 348,441 warnings in 2017
cregit+
False positive report

Fig. 4. Tracing evolution of warnings using *cregit*$^+$.

5.5. Tracing evolution of warnings

Now that we have the warnings from each of the SAST tools for the 2012 and 2017 releases of the 116 case study projects, we can trace their evolution.We track the location of each filename/line of code from the 2012 release into the 2017 release. We use the version history and *cregit*$^+$, to track the lines of code across the subsequent commits till 2017. When the 2012 line exists in 2017, we can identify if the warning is still present, has changed, or is a different one.

Therefore, by using *cregit*$^+$ we are able to determine the exact line of code in the 2017 version of the project's source code that corresponds to the line of code under consideration from 2012 (as shown in Fig. 4). We then check the warnings from 2017, to see if the determined line of code has the same warning, a different warning or no warning. Note that *cregit*$^+$ can come up with a result that the specific line of code from 2012 does not exist anymore (i.e., the line has been deleted). In this case too, there will be no warning since the line of code does not exist. We classify each of these into the 4 cases as discussed in Section 3.

Note that since we are tracing the source code across the various commits in GitHub, we only include warnings from the core project and not any third-party. Hence, we trace 213,627 warnings from 2012. All of our empirical data are available for download (Aloraini et al., 2019).

5.6. Extracting warning decay

In this step, we want to calculate the time it takes for a warning to disappear. We first included only warnings that disappear, but the lines of code have not been altered drastically which may provide more concrete results. Hence, we compare the actual line of code in 2012 and the line of code in 2017 using *Levenshtein distance* and threshold of 0.8. Then, for each warning line, we extracted the list of commits that touched that particular line. We did this using *git-blame*, for each line we found the last commit that touched it and repeated this process recursively.

6. Empirical study results

6.1. RQ1: How are the warnings from each SAST tool distributed across different warning types in 2012 and 2017?

Approach: In this RQ, we analyzed 291,794 warnings from 2012 and 348,441 warnings from 2017 in 116 projects. For comparison, we represent warning distribution using a box-and-whisker plot, with each data point representing a project. We present the plots from 2012 and 2017 side-by-side to see if the trends are changing. We present two sets of plots in this RQ. In one set, we compare the warning types and their trends between 2012 and 2017 for each SAST tool. This helps us understand if the distribution of warning types in the same tool changes between 2012 and 2017. Then, we compare the warnings across SAST tools for each warning type between 2012 and 2017. In this analysis, we are able to see which tool finds how many warnings of a particular type and if that changes between 2012 and 2017. Thus, we are able to compare different SAST tools for the same warning type.

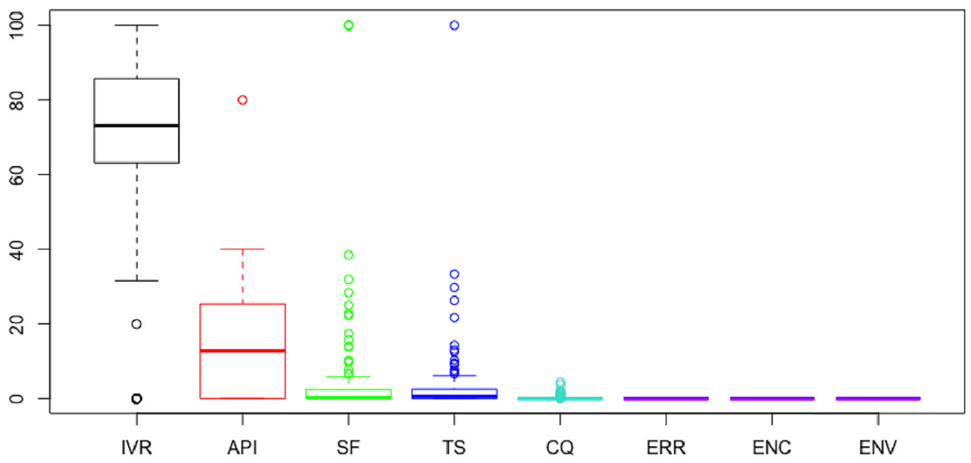
Also, we combine the box-and-whisker plot with the Scott–Knott Effect Size Difference (ESD) test to cluster different categories of warnings into statistically distinct ranks in terms of mean importance (Tantithamthavorn et al., 2017). The Scott–Knott ESD test performs well, even when the dataset involves the overlapping problem (i.e., the probability of one or more warnings to be categorized in more than one group). This test performs multiple mean comparisons, as it separates means into statistically distinct groups with non-negligible differences.

Results: The following are the results of the analysis.

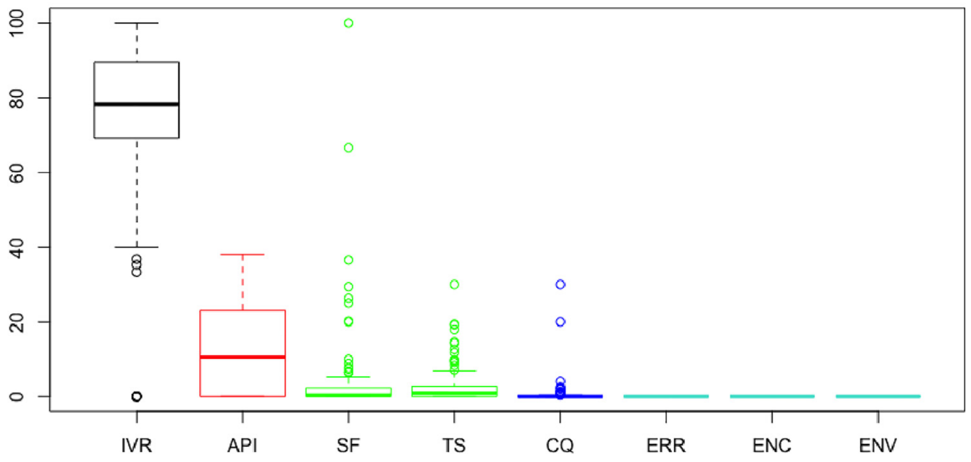
1. *Warnings within SAST tools:* Fig. 5a and b show the warnings distribution when using the RATS tool in 2012 and 2017 respectively. We include the results for the other SAST tools in Figs. 12 and 13 located in Appendix A for readability purposes. The plots include the box-and-whisker plot and the Scott–Knott ESD test among warning types ordered from left to right by their decreasing mean values. In these plots, different colors represent distinct groups that are significantly different from the other as determined by the Scott–Knott ESD test.
From the plots, we can see that the distribution of warning among different categories is almost the same in 2012 and 2017 for the RATS tool. We observe the same phenomenon in all other tools as well (check Figs. 12 and 13 located in Appendix A). Thus, the warning distributions are stable even after five years. What we can infer from these plots is that the quality of the projects from the perspective of the SAST tools are consistent. However, the different SAST tools have different vulnerability types that they detect more of.
RATS (Fig. 5a and b) and Flawfinder (Fig. 12c and d) detect more *Input Validation and Representation* (IVR) warnings than any other type of warning in both 2012 and 2017. In Cppcheck (Fig. 12e and f), the Clang Static Analyzer (Fig. 13a and b), PVS-Studio (Fig. 13c and d), and the Parasoft C/C++ test (Fig. 13e and f), we observe that *Indicator of Poor Code Quality* (CQ) warnings is the most significant type of warning that is being reported in both 2012 and 2017. Although, in all the latter tools IVR and *API Abuse* (API) warnings are the second and third most significant types of warnings being reported in both 2012 and 2017 (similar to RATS and Flawfinder).
What we can infer from the above result is that the trend of reported bug warnings for each tool does not change over time. This might indicate that the quality of projects does not change over time, which is consistent with other research findings (Edwards and Chen, 2012). Another observation is that most of the studied SAST tools produce a high number of IVR, CQ, and API warnings in both 2012 and 2017.

Takeaway 1: The trend of reported bug warnings for each tool does not change over time. This might indicate that the quality of projects does not change over time.

2. *Security warnings grouped by Seven Pernicious Kingdoms (SPK) classification:* In Fig. 6a and b we show the percentage of IVR warnings detected by the six different SAST tools in 2012 and 2017, respectively. We include the results for the other SPKs in Fig. 14 and Fig. 15 located in Appendix B. We notice that Flawfinder produces the highest number of IVR (Fig. 6a and b), API (Fig. 14c and d), and *Security Features* (SF) (Fig. 14e and f) warnings and they have the highest median among projects in both 2012 and 2017.



(a) Bug warning categories in RATS at 2012



(b) Bug warning categories in RATS at 2017

Fig. 5. Box-and-whisker plot and Scott–Knott ESD test of the number of warnings detected by RATS. In this figure, the *x*-axis represents normalized number of bugs (bugs/LOC), and the *y*-axis represents different classes of SPK bugs. Each data point represents a project. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

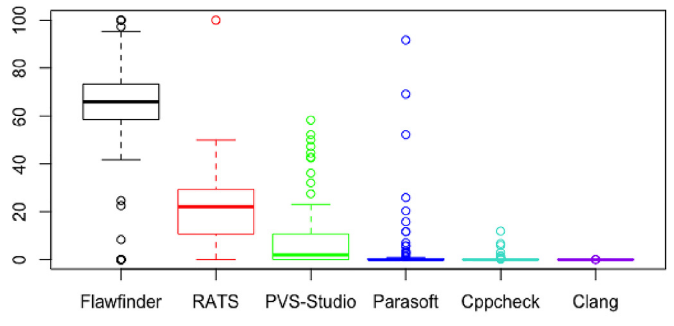
We also observe that *Time and State* (TS) results (Fig. 14g and h) is less stable between 2012 and 2017. RATS produces TS more than other tools in 2102, while Flawfinder became the most significant tool that generates TS in 2017. We also note that PVS-Studio has generated the most warnings of CQ (Fig. 15a and b), *Error Handling* (ERR) (Fig. 15c and d), and *Environment* (ENV) (Fig. 15g and h) for both 2012 and 2017. The Parasoft C/C++ test is the only tool that produced *Insufficient Encapsulation* (ENC) type of warnings (Fig. 15e and f), and it is negligible number warnings in both 2012 and 2017. The results denote that IVR, API, TS, and SF are more generated by tools that only use pattern-matching method, such as Flawfinder and RATS, while CQ is produced more by tools that use more advanced semantic analysis.

Takeaway 2: IVR, API, SF, and TS are more spotted by Flawfinder and RATS tools that use pattern-matching techniques, while CQ, ERR, ENC, and ENV are more reported by more advanced SAST tools.

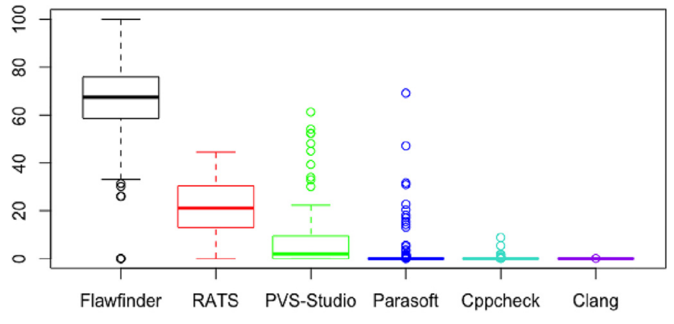
6.2. RQ2: How are true positive and false positive warnings distributed across different warning types and across different SAST tools?

Approach: We conduct our study on 213,627 warnings from the 116 projects and their versions in 2012 (which is less than the 291,794 warnings discussed in RQ1 because we only consider the core projects here and not include the dependencies like in RQ1). To answer this RQ we conducted the following analysis:

- We show the *Remained Warnings, Modified Warnings*, and *Disappeared Warnings* distributions across warning types for each SAST tool and across SAST tools for each warning type, using both box-and-whisker plot and the Scott–Knott ESD test.
- We measure the likelihood of a warning that belongs to a specific warning type to be a real warning for different SAST tools using the Odds Ratio (OR) similar to Di Penta et al. (2009). OR is the ratio of the odds of an event occurring in one set (in this research the eliminated warning subset (TP including *Modified Warnings* and *Disappeared Warnings*) to the odds of it occurring in another set (in this research the alive warning subset FP including *Remained Warnings*). An odds ratio of 1 means that the detected warnings could belong to FP or TP equally likely. An



(a) IVR warnings among SAST tools in 2012



(b) IVR warnings among SAST tools in 2017

Fig. 6. Box-and-whisker plot and Scott–Knott ESD test of the number of warnings classified as Input Validation and Representation (IVR). In this figure, the *x*-axis represents normalized number of bugs(bugs/LOC) and the *y*-axis represents different SAST tools. Each data point represents a project. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

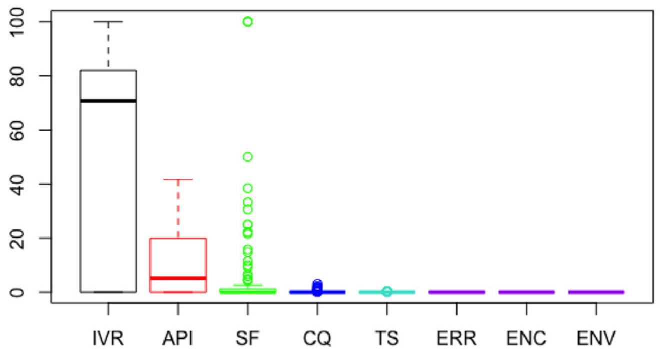
odds ratio greater than 1 means that the detected warnings are more likely to be TP, and an odds ratio less than 1 means that detected warnings are more likely to be FP.

– We apply a proportion test similar to Di Penta et al. (2009) that shows whether the proportion of eliminated warnings differ across warning types (H0: there is no difference among proportions of eliminated warnings). This could provide a better understanding of whether some types are receiving more attention than others.

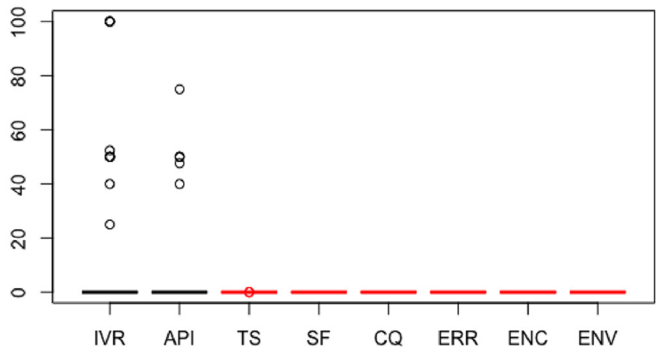
Note that, in our analysis, we only consider *Modified Warnings, Disappeared Warnings*, and *Remained Warnings*. We ignore *Removed Warnings* (one where the entire line of code from 2012 no longer exists in 2017), as this type of warning should be considered with caution since it is hard to assume that the removal of lines of code that include warnings indicates a bug fix. The line removal could be due to other reasons, such as code refactoring or deleting features.

Results: The following are the results of the analysis.

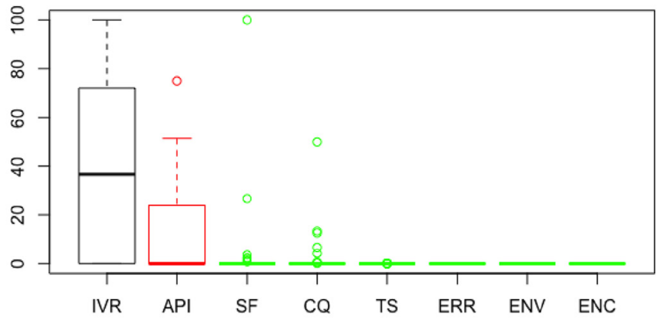
1. *Analysis of Warnings among SAST tools:* This analysis is to compare the FP and TP rates per warning types among SAST tools. Fig. 7a–c depict RATS to show *Remained Warnings, Modified Warnings*, and *Disappeared Warnings* comparisons (other plots for other tools are located in Figs. 16 and 17 in Appendix C). We observe that the *Remained Warnings* plots have a similar pattern to the *Disappeared Warnings* plot for each tool. RATS (Fig. 7a–c) and Flawfinder (Fig. 16d–f) show that the most significant groups of *Remained Warnings* and *Disappeared Warnings* are IVR, API, and SF, respectively. In Cppcheck (Fig. 16g–i), PVS-Studio (Fig. 17d–f), the Clang Static Analyzer (Fig. 17a–c), and the Parasoft C/C++ test (Fig. 17g–i), we note that CQ is the most types of warnings that appear as *Remained Warnings* and *Disappeared Warnings*. It is remarkable to see that in RATS, Flawfinder, and the Parasoft C/C++ test have some warnings in *Modified Warnings* plots, which refers to the warnings that likely have been resolved but there were other warnings have been introduced. An example of that is when developers try to replace possibly insecure function *strcpy* by another more secure but still vulnerable function *strncpy*. Hence, we infer from the above results that there is a strong correlation between the number of FP and TP for each type of warning in each tool. For instance, the most frequent type of false positive bug warnings produced by RATS tool is IVR, and also IVR is the most frequent type of bug warnings that disappeared. This means that the number of false positive could be used as an indication of the potential real bugs could be detected by each tool.



(a) *Remained Warnings* in RATS

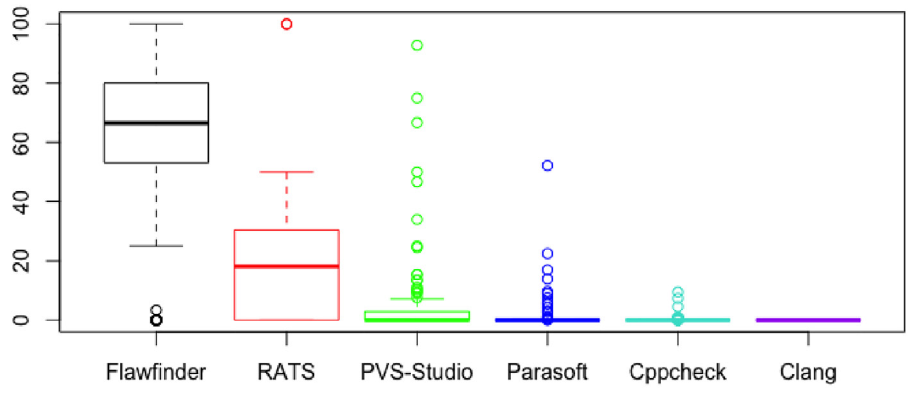


(b) *Modified Warnings* in RATS

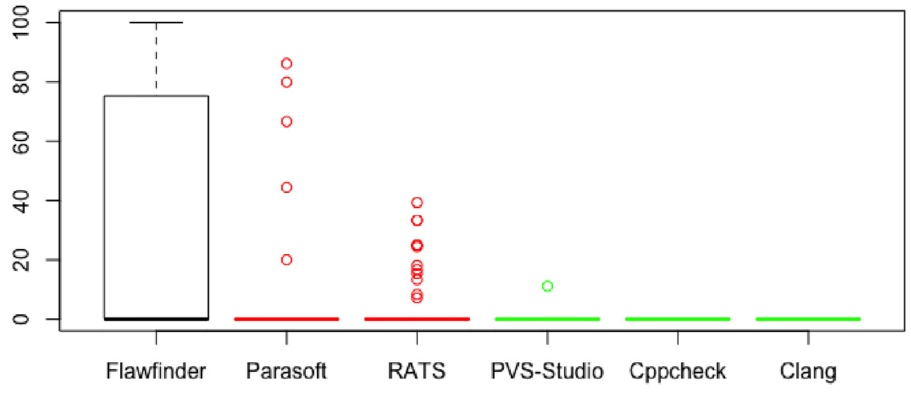


(c) *Disappeared Warnings* in RATS

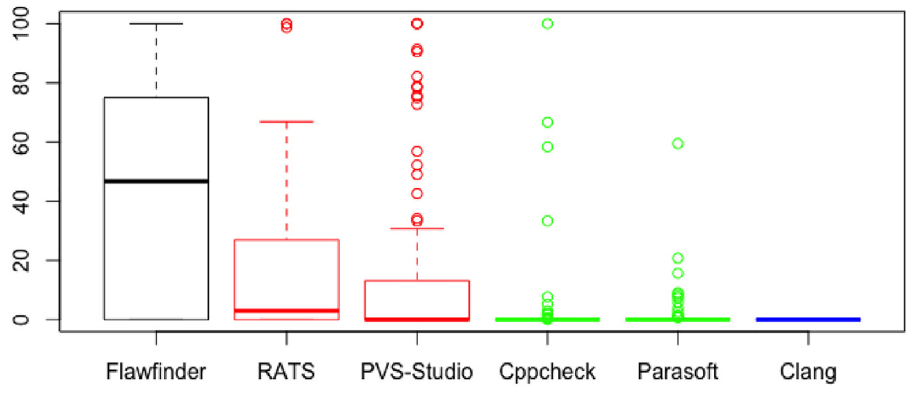
Fig. 7. Box-and-whisker plot and Scott–Knott ESD test of the *Remained Warnings* (FP), *Modified Warnings* and *Disappeared Warnings* (TP) among different warning types by RATS. In this figure, the *x*-axis represents normalized number of bugs(bugs/LOC), and the *y*-axis represents different classes of SPK bugs. Each data point represents a project. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



(a) *Remained Warnings* among IVR



(b) *Modified Warnings* among IVR



(c) *Disappeared Warnings* among IVR

Fig. 8. Box-and-whisker plot and Scott–Knott ESD test of the *Remained Warnings* (FP), *Modified Warnings* and *Disappeared Warnings* (TP) among different warning types among Input Validation and Representation (IVR). In this figure, the *x*-axis represents normalized number of bug(bugs/LOC), and the *y*-axis represents different SAST tools. Each data point represents a project. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

> **Takeaway 3:** For all the SAST tools used in this study, for each type of warning there is a strong correlation between the number of true positive and the number of false positive warnings found.

2. *Analysis of Warnings among SPKs:* Another analysis we performed is to compare each type of warnings among different SAST tools. Fig. 8a–c display IVR warnings to show *Remained Warnings, Modified Warnings*, and *Disappeared Warnings* rate comparisons (while other plots for other tools are located in Figs. 18 and 19 in Appendix D). In IVR (Fig. 8a–c), API (Fig. 18d–f), and SF (Fig. 18g–i) plots, we see that Flawfinder produces a significant amount of *Remained Warnings*; in addition, it has the most significant number of warnings detected correctly in *Disappeared Warnings*. RATS is the most tool that produces TS as both *Remained Warnings* and *Disappeared Warnings* (Fig. 18j–l), while PVS-Studio is the most tools that produces CQ (Fig. 19a–c) and ERR (Fig. 19d–f) as *Remained Warnings* and *Disappeared*

Table 4
Numbers of all detected FP (which includes *Remained Warnings*), TP (which includes *Modified Warnings* and *Disappeared Warnings*), and Odds Ratio of with respect to FP.

Tool	Type	Total	Remained warning	Modified warning	Disappeared warning	TP	TP+FP	Removed warning	OR
RATS	IVR	37,776	26438(69.99%)	84(0.22%)	3681(9.74%)	3765	30,203	7573	0.020
	API	10979	6770(61.66%)	80(0.73%)	1772(16.14%)	1852	8622	2357	0.074
	SF	592	409(69.09%)	0	46(7.77%)	46	455	137	0.012
	CQ	157	104(66.24%)	0	25(15.92%)	25	129	28	0.057
	TS	1246	827 (66.37%)	3(0.24%)	140(11.24%)	143	970	276	0.029
Flawfinder	IVR	92,998	68126(73.26%)	320(0.34%)	5992(6.44%)	6312	74,438	18,560	0.008
	API	35,314	24491(69.35%)	234(0.66%)	2731(7.73%)	2965	27,456	7858	0.014
	SF	1157	882(76.23%)	4(0.35%)	48(4.15%)	52	934	223	0.003
	CQ	3250	2801(86.18%)	8(0.25%)	115(3.54%)	123	2924	326	0.001
	TS	2168	1315(60.65%)	5(0.23%)	144(6.64%)	149	1464	704	0.012
Cppcheck	IVR	274	144(52.55%)	0	41(14.96%)	41	185	89	0.081
	API	207	115(55.56%)	0	30(14.49%)	30	145	62	0.068
	CQ	1563	850(54.38%)	1	385(24.63%)	386	1236	327	0.206
Clang Static Analyzer	IVR	1	0	0	0	0	0	1	-
	CQ	31	14(45.16%)	0	5(16.13%)	5	19	12	0.127
PVS-Studio	IVR	7154	2408(33.66%)	1(0.01%)	3461(48.38%)	3462	5870	1284	2.067
	API	1656	635(38.35%)	0	679(41.00%)	679	1314	342	1.143
	CQ	16,808	6295(37.45%)	9(0.05%)	7464(44.41%)	7473	13,768	3040	1.409
	ERR	208	50(24.04%)	0	83(39.90%)	83	133	75	2.755
	ENV	29	16(55.17%)	0	11(37.93%)	11	27	2	0.472
Parasoft C/C++ test	IVR	1835	961(52.4%)	152(8.3%)	658(35.9%)	810	1771	64	0.710
	API	1792	1010(56.4%)	158(8.8%)	554(30.9%)	712	1722	70	0.496
	SF	5	1(20.0%)	0	4(80.0%)	4	5	0	16
	CQ	78322	42541(54.3%)	1790(2.3%)	32764(41.8%)	34554	77095	1227	0.659
	TS	156	101(64.7%)	18(12.8%)	32(19.2%)	50	151	5	0.245
	ENC	2	2(100%)	0	0	0	2	0	-
	ENV	39	12(30.8%)	0	21(53.8%)	21	33	6	3.062

Warnings. We note that ENC (Fig. 19g–i) are produced only by the Parasoft C/C++ test, but it is likely to be *Remained Warnings* that are FP warnings. Similarly, the results indicate that a tool that generates the highest number of false positive of a specific warning type compared to other tools, it would be the tool that the most disappeared warnings of the same type belong to. For instance, Flawfinder generates the most false positive IVR bug warnings compared to other tools, also the disappeared warnings of the IVR mostly belong to Flawfinder. This also could be used as an assessment to measure the potential real bugs of a certain type could be detected by such SAST tool.

Takeaway 4: Pattern-matching based tools, such as RATS and Flawfinder, produce a larger number of warnings (both FP and TP) of IVR, API, SF, and TS, while more advance tools, such as PVS-Studio and Parasoft C/C++ test, generate a larger number of CQ warnings (both FP and TP).

3. *Odds ratio analysis:* Table 4 reports all SAST-produced warnings in terms of:
   - The total number of warnings reported by each tool split by warning type in 2012.
   - The number of warnings which exists in the same line of code in both the 2012 and 2017 versions (*Remained Warnings*).
   - The number of warnings from 2012 which disappear in 2017, but have a different warning in the same line of code as compared to 2012 (*Modified Warnings*).
   - The number of warnings from 2012 that no longer exist in the same line of code, but the line of code still exists, albeit modified (*Disappeared Warnings*).
   - The odds ratio of a tool finding a warning in 2012.

   RATS, Flawfinder, Cppcheck, and Clang Static Analyzer have Odds Ratio (OR) less than one for all warning types, indicating that these types of warnings are likely to be FP.
   Conversely, PVS-Studio results denote that most reported warnings in 2012 disappeared in 2017, hence they are more likely to be real warnings. Only ENV type of warnings is the most likely to be FP. Parasoft C/C++ test is likely to be accurate with SF warnings. We note that it has (OR=16), which is a very high odds ratio to be a real warning compared to other types of warnings. Similarly, the ENV type of warnings show (OR=3.062), which indicates a higher probability of having real warnings reported. That is because Parasoft C/C++ test enforce secure coding standards, such as MISRA, CERT, and ISO. Also, Parasoft C/C++ test detects Environment errors with high precision rate because it targets cross-compiler platforms.
   From the result above we note that tools that use pattern-matching along with other advanced analysis methods, such as PVS-Studio and Parasoft C/C++ test, outperform other tools in term of the likelihood of producing real security bugs.

Takeaway 5: SAST tools in this study that use pattern-matching method along with semantic analysis are more likely to produce warnings that may be real warnings.

4. *Proportion test analysis:* For warnings detected with RATS and Flawfinder, the proportion test of likely resolved warnings significantly varies across types (p-value < 0.0001), meaning that there are some types receiving more attention and likely being resolved than others. A small p-value (typically ≤ 0.05) indicates strong evidence against the null hypothesis, so we could reject the null hypothesis, which states that there is no difference among proportions. For warning detected with Cppcheck, the proportion of likely fixed warnings slightly varies

across types (p-value = 0.0025), which denotes that there is a difference in the attention paid to different warning types detected by this tool. When looking at PVS-Studio results, we observe that the proportion test of likely fixed warnings significantly varies across types (p-value = 4.008e−10). Finally, when glancing at the Parasoft C/C++ test, we find that the proportion test of likely fixed warnings (TP) varies slightly across types (p-value = 0.0002). This denotes that there is a difference in the attention paid to different warning types detected by this tool. Thus, we conclude that for each studied SAST tool there are some types of warnings that receive more attention from the project teams to be eliminated, while other types of warnings are likely are ignored suggesting that they are not critical bugs.

Takeaway 6: For every SAST tool, some types of warnings were more likely to be eliminated than other types. Hence, it is very likely that some types of warnings receive more attention from developers.

6.3. RQ3: How do different types of warnings evolve over time?

Approach: In this RQ, we focus on TP warnings (*Modified Warnings* and *Disappeared Warnings*) that were detected by SAST tools and eliminated in the code. We only consider lines of code that include warnings and have been altered, as this gives us the ability to trace the changes in the lines of code. Whenever we find a warning that was eliminated and belong to a line of code that was altered in the recent version, we trace the line of code that contains the bug warning backward during the history to find the fix time and flag that to be a *bug fix* version. To answer this question, we use *git blame*. This study involves 10,478 warnings that have been eliminated by altering the lines of code. We present the results using box-and-whisker plots. In these plots each data point is how long it took for a specific warning in a specific line of code from one of the 116 projects, to disappear.

Results: The following are the results of the analysis.

1. *Warnings decay among SAST tools:* We analyze the set of *Disappeared Warnings* and *Modified Warnings* that present in 2012 and the lines of code that were after being altered the warning disappeared. This helps us to measure the decay. Fig. 9 shows the decays (expressed in days) for each warning type detected in the projects with different SAST tools. When looking at the plots, we observe that most of the warnings that belong to different types eliminated approximately within 2–3 years. We observe also that in Flawfinder plot, the median of API decay is significantly lower than the other warning types (approximately less than one year) suggesting that there is a tendency to eliminate this type of warnings faster than others. Generally, we observe that API warnings disappeared relatively faster than other types among tools. In addition, the Parasoft C/C++ test plot shows that TS warnings are eliminated relatively fast, indicating that this type of warnings has less removal time, which may indicate that this type of warnings is more critical.

Takeaway 7: Most of the bug warnings that belong to different types disappeared approximately within 2–3 years. Generally, API warnings disappeared relatively faster than other types among tools.

2. *Warnings decay time among SPKs:* Here, we are interested in knowing if a certain type of bug warnings could be eliminated faster if it is detected by a certain SAST tool. This allows us to know if the underlying analysis method influences the removal time of a bug warning of a certain category. Note that tools that only use pattern-matching method would refer to an issue that is seen clearly within the same line of code, while advanced tools could comprehend errors that are scattered on multiple lines and thus may be harder to be understood. Fig. 10 shows the results of such an analysis. The results imply that for IVR and CQ, the removal time is almost the same among all tools. Also, it is interesting to note that IVR removal time for tools that are only use pattern-matching analysis is slightly less than other tools that use more advanced techniques. This could be because these tools generated trivial bug warnings that are easier to understand and resolve. We perceive also that generated API warnings by Flawfinder decay less than other API warnings generated by other tools (in less than one year). Furthermore, we see that TS generated by the Parasoft C/C++ test decay in approximately one year. Another remarkable observation is that SF type of warnings usually takes more than three years to be eliminated and this is longer than any other warning type detected by most of the analysis tools. We note that there is no connection between decay time and different SAST tools that have different analysis methods. Hence, the nature of the bug warning may not be a factor that influences removal time.

Takeaway 8: There is no correlation between decay time and different SAST tools that have different analysis methods.

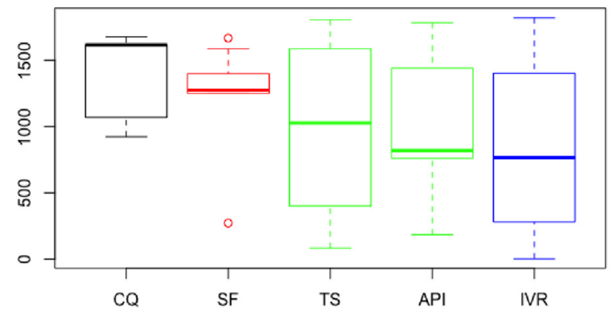
7. Discussion

7.1. SAST warning patterns as and real-world vulnerabilities

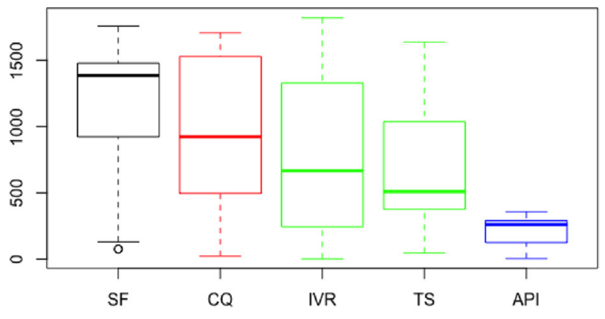
When observing the results from our analysis, we notice that SAST-produced warnings have stable patterns through time (check RQ1 in Section 6). This may indicate that the quality of the project's development does not change over time. In addition, we find that most of the SAST tools focus on IVR, API, and CQ types of warnings. This may denote that these types of warnings are critical and could be the types of security vulnerabilities that occur in real-world. Real-world vulnerabilities are exploitable software bugs that have been discovered and published in recognized vulnerability databases with unique identification numbers. To verify this, we study the distribution of real-world vulnerabilities and associated vulnerability classifications in the studied projects that are fixed by the project developers, to see if this aligned with our findings. So, we conduct the following analysis:

7.1.1. Identifying incidence of real-World vulnerabilities

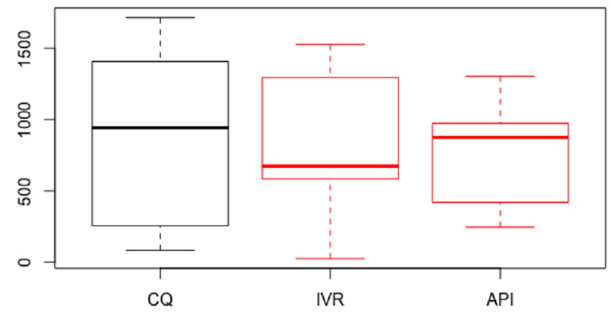
We choose to study vulnerabilities that have a Common Vulnerability and Exposures Identification Number (CVE-ID), in the 116 projects in our dataset. The CVE-ID are unique numbers assigned to publicly known security vulnerabilities. In this step, we extract all commit messages along with the description of each commit looking for the keyword “CVE-xxxx-xxxx” between 2012 and 2017. We find that of the 1174 times a CVE-ID was mentioned in 18 distinct projects, 395 have a unique ID. This helps us to identify how real-world vulnerabilities are distributed across various types of vulnerabilities.



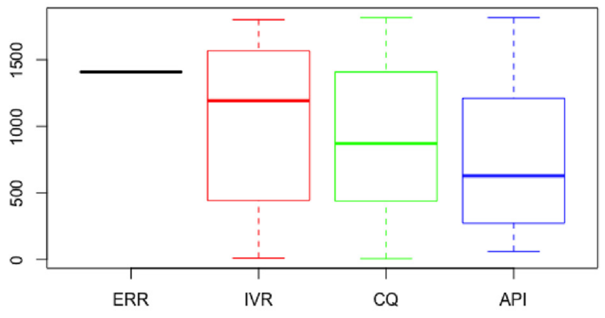
(a) Decay in RATS



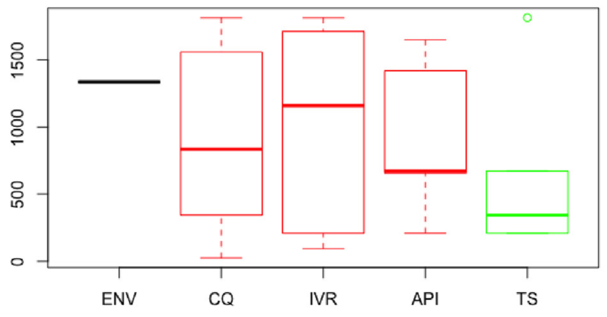
(b) Decay in Flawfinder



(c) Decay in Cppcheck



(d) Decay in PVS-Studio



(e) Decay in the Parasoft C/C++test

Fig. 9. Box-and-whisker plot of decays for various warning types in different SAST tools. In this figure, the *x*-axis represents decays in days, and the *y*-axis represents different classes of SPK warnings. Each data point represents a resolved bug warning time. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

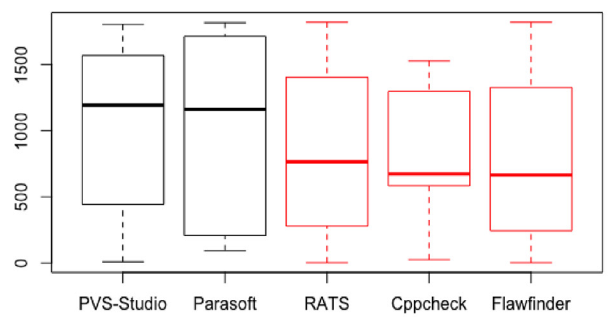
7.1.2. Results of real-world vulnerabilities distribution

Fig. 11 displays the results of real-world vulnerabilities that have been fixed in the projects under the study. We can see that the IVR vulnerabilities are the predominant type of vulnerability that has been detected and fixed in real-world scenarios. The number of IVR vulnerabilities is 293. The other types of vulnerabilities that are given importance after IVR are CQ, API, and SF, respectively.

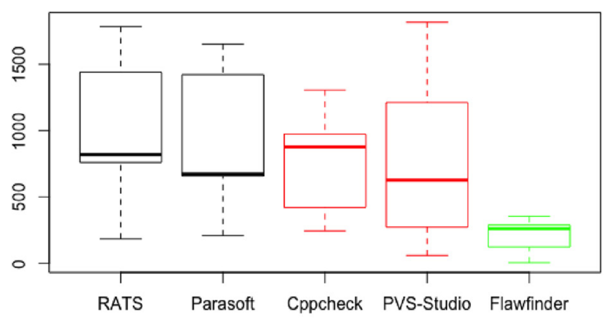
The results show that IVR is the most significant type of warnings being addressed in real-world vulnerabilities. Most of the studied tools produce a high number of IVR warnings. This may refer to the importance of this type of warning and hence the importance of enhancing the static analysis methods to accurately detect IVR. Also, CQ and API issues were being flagged in real-world, similarly to the output of SAST tools. Generally, we conclude that SAST tool warnings follow a similar distribution of the discovered real-world vulnerabilities in the studied projects, which is a finding that is consistent with other research (Edwards and Chen, 2012).

7.2. Robustness of static analysis method

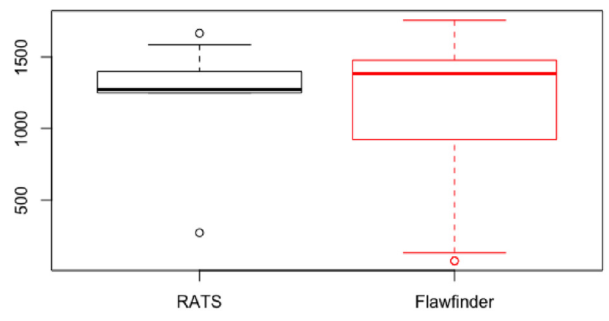
The results of RQ2 in Section 6 highlight the odds of each type of warning of being true or false alarms for each tool. The results indicate that PVS-Studio and the Parasoft C/C++ test produce warnings that are likely to be real warnings. PVS-Studio has higher chances to produce real IVR, API, CQ, and ERR, while the Parasoft C/C++ test has higher chances to produce real SF and ENV. These two tools are the only tool who combine pattern-matching with more advanced semantic analysis (see Table 1 for more details). In fact, pattern-matching by it is own is a powerful method that can detect many real bugs. That is notable since Flawfinder and RATS outperform PVS-Studio in finding more IVR warnings, and Flawfinder outperforms PVS-studio in finding more API warnings in terms of the frequency/quantity. This denotes the value of using the pattern-matching method since pattern-matching tools (Flawfinder and RATS) could detect more real bugs (quantitatively not qualitatively). For example, Flawfinder detected 5992 of In-



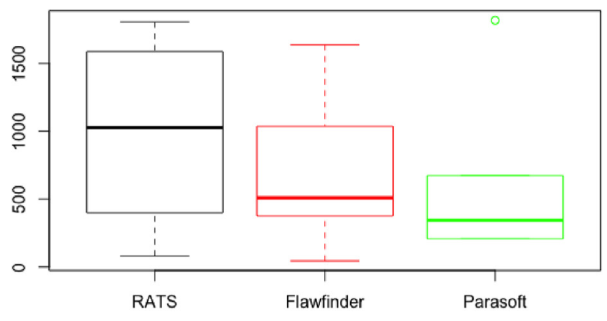
(a) Decay of IVR



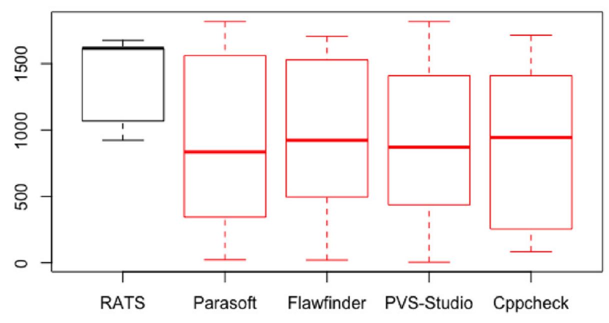
(b) Decay of API



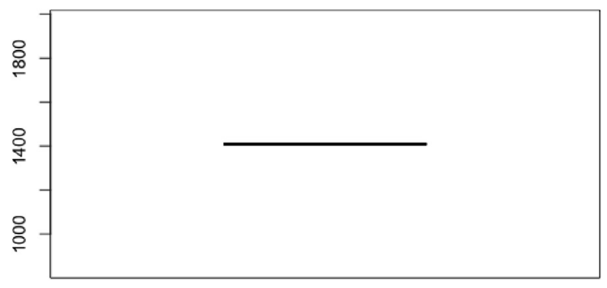
(c) Decay of SF



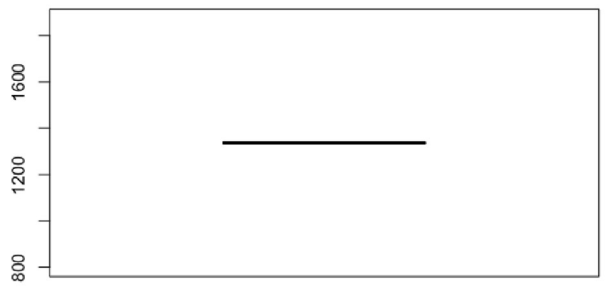
(d) Decay of TS



(e) Decay of CQ



(f) Decay of ERR



(g) Decay of ENV

Fig. 10. Box-and-whisker plot of decays for various warning types in different warning types. In this figure, the *x*-axis represents decays in days, and the *y*-axis represents different SAST tools. Each data point represents a resolved bug warning time. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

put validation and representation bugs while PVS-Studio 3461 bugs of the same type. Yet, Flawfinder has around 6.5% accuracy while the PVS-Studio rate is 48% which is better qualitatively. However, when combining pattern-matching with other analysis techniques, such as symbolic execution or abstract interpretation, the accuracy of the warning increases. Hence, we think that pattern-matching is a robust technique but it needs to be combined with a more advanced analysis method to filter out any possible false positives. Also, ENV warnings are likely to be accurately flagged by the Parasoft C/C++ test than PVS-Studio. This is because that the

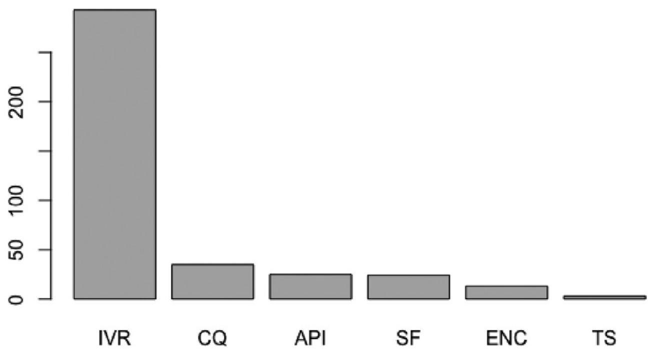


Fig. 11. CVE vulnerabilities distribution based on the SPK taxonomy. In this figure, the *x*-axis represents the number of unique vulnerabilities and the *y*-axis represents different classes of SPK classification of bugs.

Parasoft C/C++ test highly focused to highlight software environmental compliance.

7.3. SPK warnings decay time

In RQ3 Section 6, we observe that of those warnings that are removed, the decay time is around 2–3 years. This result corresponds to other research findings, such as Ozment and Schechter (2006). Ozment and Schechter (2006) study the vulnerabilities of OpenBSD operating system to investigate whether the software vulnerabilities are increasing over time. The authors measured the rate of reported vulnerabilities over a period of seven and a half years that span 15 releases. The study showed that vulnerabilities seem to be persistent even for a period of 2.6 years. Also, among studied tools, we find that generally, API has decay somehow faster compared to other types of warnings. This may imply the significance of such a warning that may have serious consequences. This could be helpful to give recommendations to developers about which bugs should be fixed first. Another possible reason is that this is because these types of bugs are easier to understand and fix since the API issue is likely to manifest in the same line of code compared to other types of bugs that could be caused by a different line of code. For instance, a buffer overflow warning that belongs to IVR class of bug could happen when a condition that does not validate the size of buffer located in a different line of code. However, this is less likely since we have a limitation in RQ3 that we only examine warnings that disappear because the line of code has been modified.

8. Lessons learned

In this section, we provide insights about how the findings can improve the state of the art and the state of the practice for developers, security testers, SAST tool designers, and researchers.

– **Software developers:** Building secure software in the first place is key to prevent security bugs from ever occurring. This not only requires solid knowledge about secure coding practice but also building practical skills that can be learned and built from real-world problems. Although developers are expected to write secure code, around 70% of developers said they get little guidance or assistance according to a survey (Thornburg, 2019). This paper helps developers to be aware of common security bugs that are regularly addressed. The paper shows that IVR issues are the most types of vulnerabilities that have been fixed in real-world projects. Also, this work demonstrates that most of the SAST tools focus on IVR, CQ, and API types of issues. This may indicate that these types of warnings are critical. Hence, it would be advisable that developers build related secure coding practice and be aware of that during coding and take the time to ensure the code is free from these types of bugs.
Additionally, SAST tools could generate a large number of warnings that could overwhelm the most experienced software developers. This paper shows that some types of warnings are not likely to be removed, hence they tend to be false positives. Code that seems to be reliable and working well contains warnings, hence aiming to remove all warnings might not be cost effective.
Furthermore, one of the implications of this study is that developers need to pay closer attention to the SAST tools appropriate underlying design to detect a certain type of bug. For instance, for developers who develop web applications and interested in detecting IVR type of bug, it would be better to use a tool that uses both pattern-matching and symbolic execution which was shown in this work to have a higher chance that generated warnings that are likely to be a real vulnerability.
– **Software security testers:** Software testers are responsible for investigating and evaluating the security and quality of software components. Running SAST tools may produce many warnings for a software component, and developers do not care to remove them all. Therefore, it is important for software testers to be able to prioritize bug warning categories into importance, so when software testers scan the code, they should look for those that are more likely to create bugs. In addition, software testers usually need to provide early estimates and an objective view of the software reliability or fault-proneness to help code inspections and testing and understanding the risks of software implementation. The empirical evidence in this research (RQ2) shows that using SAST tools might be a suitable method to measure the quality of the project despite the false positive rates. Therefore, SAST tool warnings can predict the security risk of the products even without a manual review of the warnings. Hence, false warnings are not completely unbeneficial since we observe that the more false positive warnings of a certain type by a SAST tool the more real bugs being fixed (see RQ2 in Section 6). Also, this study demonstrates that security bugs seem to be persistent even for a period of 2.6 years, which is consistent with previous research studies (Ozment and Schechter, 2006). This means that when assessing a bug report, it should be taken into consideration that some critical bug's lifetime can be long and not to neglect it for that reason.
– **SAST tool designers:** The take home message for SAST tool designers is that not all types of warnings appear to be critical to developers. This paper shows that some types of warnings are never removed, then the challenge is to identify or rank warnings according to the likelihood that they will be a bug. While this is ideal, the work herein provides an easier-to-implement alternative: rank the importance of warnings based on whether they will be removed in the future. While this is imperfect, this might be better than no ranking at all. Moreover, this paper finds in RQ2 that pattern-matching is a robust technique but it needs to be combined with a more advanced analysis method to filter out any possible false positives. We also noticed that abstract interpretation can be an optimal approach to design tools that focus on Security Features (FS) security bugs, such as a privacy violation.
Another point to be highlighted is that by observing that API type of bugs decay faster than other types and underlying analysis techniques does not influence that. We think that these types of bugs were fixed faster because once identified, it was easier to understand and hence fixed. In fact, previous studies have shown that the difficulty of understanding a bug warning may lead to neglecting it (Johnson et al., 2013). Unlike API bug that could manifest on one line of code, other types of bug warning can include issues that are scattered across multiple lines of code. Hence, bug warnings need to be represented well to developers. For instance, instead of showing a line of code

that includes a bug and explaining that abstractly. Tool designers need to illustrate the set of lines of code that involve the problem.

– **Researchers:** This paper illustrates that some types of warnings are more likely to disappear. Hence, research is really needed into trying to understand if this is indeed because either: developers concentrate on these types of errors (they are low hanging fruit during code reviews and other when they edit the code around) or because they indeed create bugs and the warning indeed show the cause of the problem. Another research direction worthy of investigation is to comprehend the reasons for short-lifetime of bug warnings, as well as incorrect fixes and buggy patches. Also, researchers could benefit from this research as they can continue this work and try to improve it by tracking more data/better data, surveys, more systems, etc.

9. Threats to validity

Threats to construct validity: In this study, we are not assuming that developers have run the SAST tools in this paper. We only measure the issue with the source code that could be flagged by SAST tools and measure the FP and TP rate by observing the change in the line of code. We make the following assumptions: (a) If a warning exists in the released version of the project in 2012 and 2017, then that warning is a false positive warning. This is a fairly reasonable assumption since any warnings that are present in the released code for five years are likely do not affect the quality of the product. Hence, they are FP. (b) If a warning no longer exists in the 2017 version, we assume that it is a TP. We do not know why the developer resolved the warning, but just that the warning no longer exists. Irrespective of the reason, since the developers chose to take some action that resulted in the warning to go away, we assume that this is important. However, the developers may not remove the warnings consciously. Therefore, this could be a problem. One action we took to address this is that we do not discuss *Removed Warnings* (when a line is deleted) in our paper. We believe that removing lines could be because of removing a feature. While editing a line is most likely to fix a bug or for maintenance purposes. Hence, we only discuss *Modified Warnings* and *Disappeared Warnings*.

Additionally, we manually reviewed some bug warnings to mitigate the threats to validity of our results regarding 1) the false positives and true positives rates of the same warning types that vary across tools, 2) decay time of bug removal of the same warning types that differ across tools. Different tools produce different outputs in terms of false positive and true positive rates for the same warning type. This is because different tools have different underlying analysis techniques (as discussed in Section 5.2). Hence, IVR warnings generated by RATS, which is based on pattern-matching, differ from IVR warnings generated by PVS-Studio, which is based on pattern-matching and symbolic execution. However, our results show that the underlying analysis method does not appear to influence how fast the bug is being fixed. Hence, to ensure that our results are concrete we manually looked at a few cases of Modified Warnings and Disappeared Warnings to see if the developers indeed removed the warnings. We manually analyzed 40 warnings, by reviewing the commit notes and source code, and we found that 28 cases of the fixed commits indicate bug fixing tasks. In fact, 11 commits out of 28 indicate that the bugs were detected by SAST tools, such as PVS-Studio and Cppchek among other tools. For instance, a warning that belongs to InsightSoftwareConsortium/ITK project has disappeared; the fixing commit a3793658d2 (i.e., the commit that removed the warnings as identified in RQ3) shows this commit message “COMP: Fix all valid cppcheck warnings in ITK (last patch)”. While other commits do not have enough information about bugs being fixed. In this case, we notice that the source code includes many changes that are more important in those versions, such as adding features which are project dependent. In fact, this issue was observed in Kim and Ernst work (Kim and Ernst, 2007) (check Section 2), when the researchers found that bugs that were fixed in fix-changes represent a very small percentage of bug removals, and about 90% of the bug warnings either remain in the program or are fixed in non-fix changes.

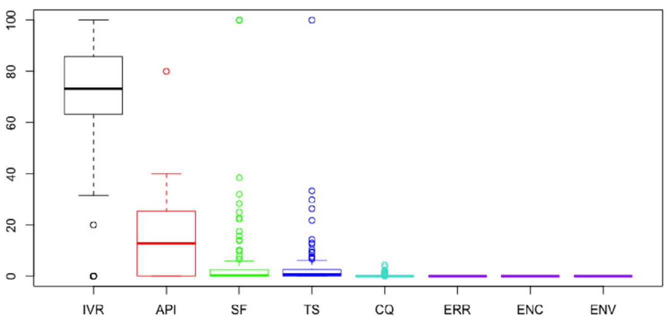
Threats to internal validity concern factors that could have affected our findings. We ensure that the selected projects do not introduce any biased results to our study. We study 116 popular C++ projects, yet they vary in terms of complexity, size, and activeness. Also, when we mined the chosen repositories from GitHub, we enforce some criteria to ensure the robustness of our dataset. For instance, we built a script to ensure that retrieved projects are C++ projects, and not falsely tagged in Github as C++. Another threat is that we chose an analysis time frame of five years (2012–2017). From past research, we know that the typical time to fix a bug is around three years (Canfora et al., 2009; Di Penta et al., 2009). Hence, we considered a five-year time frame so that we can be sure that if a warning existed in both versions, then it has a life span that is at least five years. Finally, we provide all our tools and data for replication (Aloraini et al., 2019), and the methodology of this study is described in details in Section 5.

Threats to external validity This study was conducted on 116 different open source projects and six different SAST tools. Yet, analyzing further projects, different programming languages, and other SAST that have different analysis techniques are desirable.

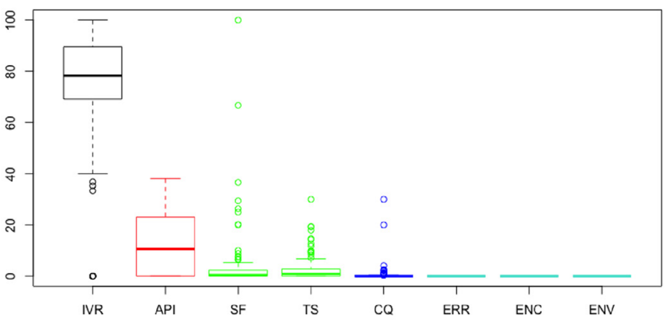
10. Conclusion

This work provides an empirical study to the warnings generated by SAST tools and investigates the history of these buggy lines of code, using 116 large C++ repositories. We find that the patterns of the warnings are stable through time for all studied SAST tools. Also, our results show that most of the tools produce on IVR, API, and CQ warnings. These types of warnings are detected in real-world with comparable patterns, but IVR is detected significantly in real-world compared to others. Also, we observe that there is a correlation between the number of bugs belong to different bug types generated by such tool and the likely real bugs that were resolved. This indicates that SAST tools could be used as an assessment tool to measure the quality of a product and the potential risks without manually review the warnings. Also, this study shows the power of using pattern-matching algorithm along with other advanced analysis methods. The outcome of this research may have multiple possible future directions, such as bug warnings prioritizing. Knowing the relation between static analysis methods and false alarm could help in designing better SAST tools to minimize false alarm rates. Moreover, it provides an insight into which static analysis algorithm is appropriate for different classes of warnings.

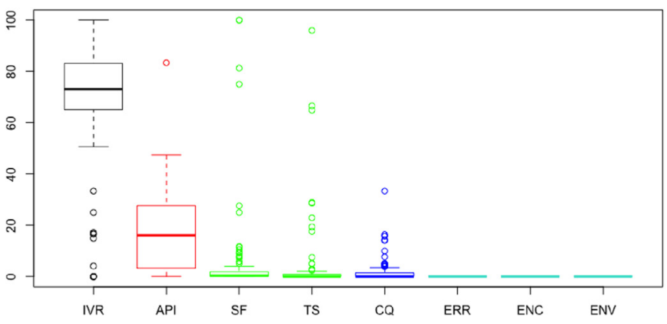
Appendix A. Bug warning distribution in 2012 and 2017 among SAST tools (RQ1)



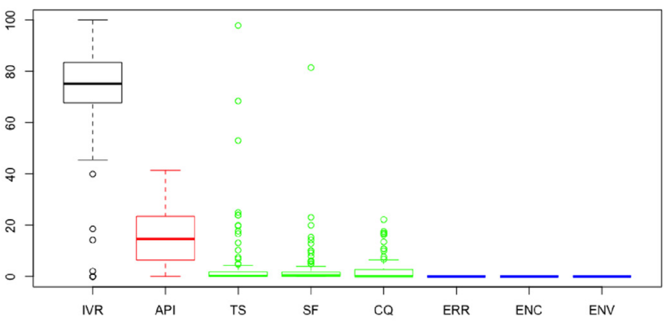
(a) Bug warning categories in RATS at 2012



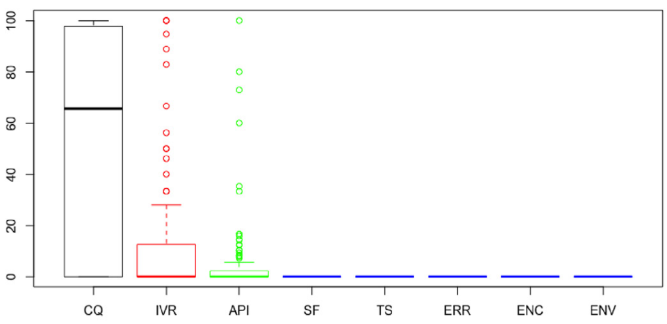
(b) Bug warning categories in RATS at 2017



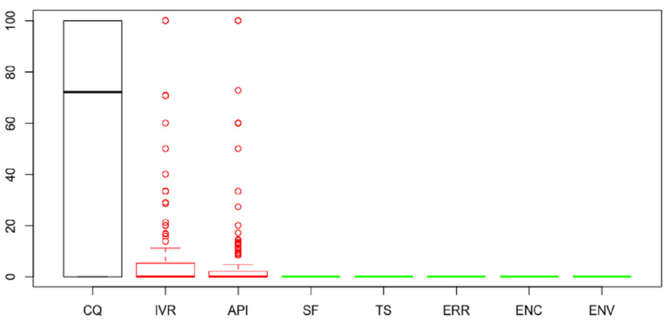
(c) Bug warning categories in Flawfinder at 2012



(d) Bug warning categories in Flawfinder at 2017

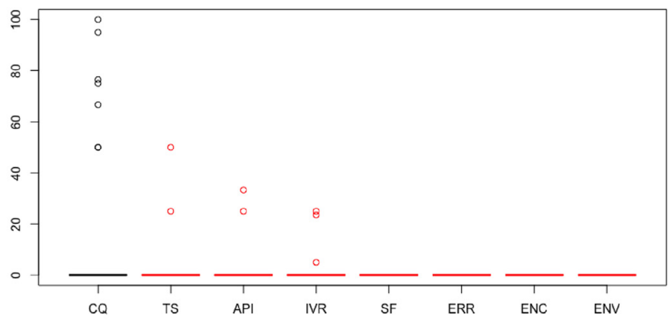


(e) Bug warning categories in Cppcheck at 2012

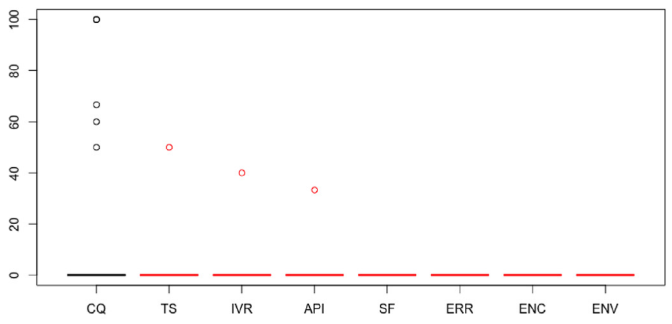


(f) Bug warning categories in Cppcheck at 2017

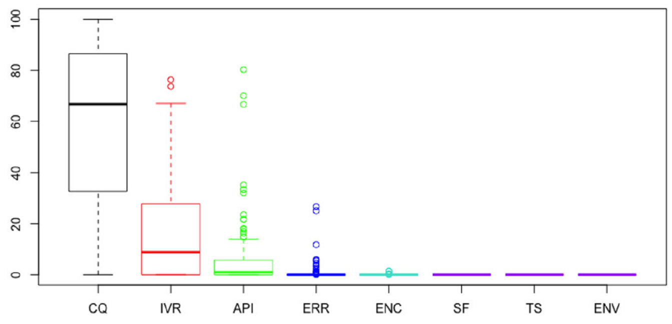
Fig. 12. Box-and-whisker plot and Scott–Knott ESD of the number of bugs detected by RATS, Flawfinder, and Cppcheck. In this figure, the *x*-axis represents normalized number of bug(bugs/LOC), and the *y*-axis represents different classes of SPK bugs. Each data point represents a project. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



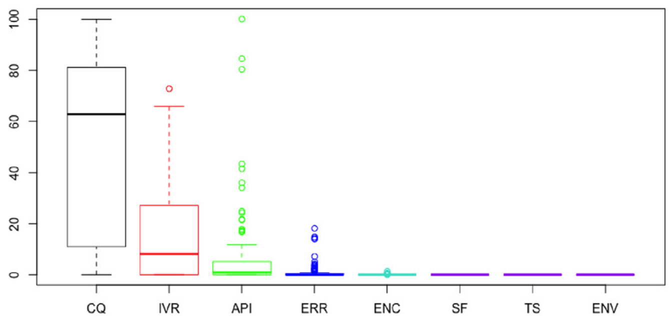
(a) Bug warning categories in the Clang Static Analyzer at 2012



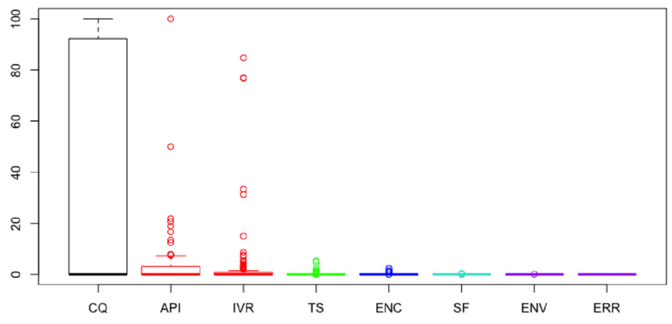
(b) Bug warning categories in the Clang Static Analyzer at 2017



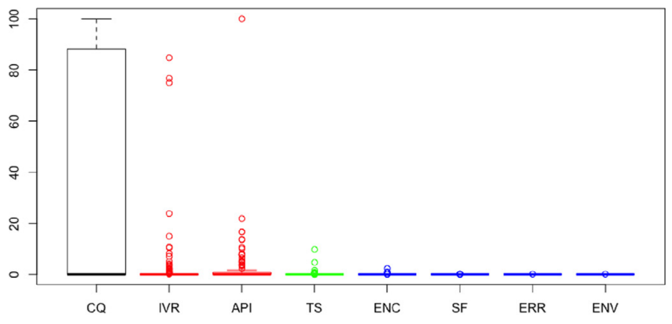
(c) Bug warning categories in the PVS-Studio at 2012



(d) Bug warning categories in the PVS-Studio at 2017



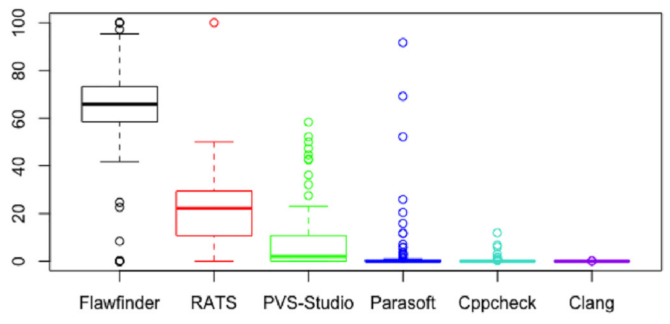
(e) Bug warning categories in the Parasoft C/C++test at 2012



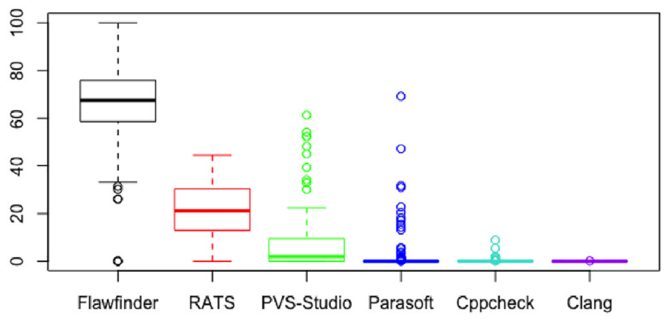
(f) Bug warning categories in the Parasoft C/C++test at 2017

Fig. 13. Box-and-whisker plot and Scott–Knott ESD of the number of bugs detected by Clang Static Analyzer, PVS-Studio, and Parasoft C/C++ test. In this figure, the *x*-axis represents normalized number of bug(bugs/LOC), and the *y*-axis represents different classes of SPK bugs. Each data point represents a project. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

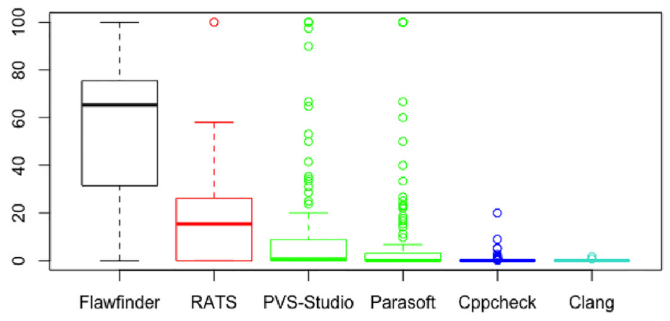
Appendix B. Bug warning distribution in 2012 and 2017 among SPK categories (RQ1)



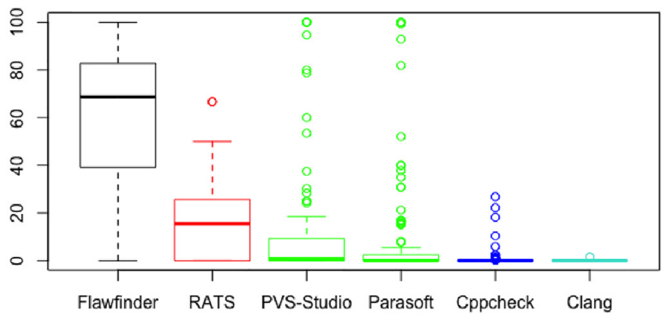
(a) IVR warnings among SAST tools in 2012



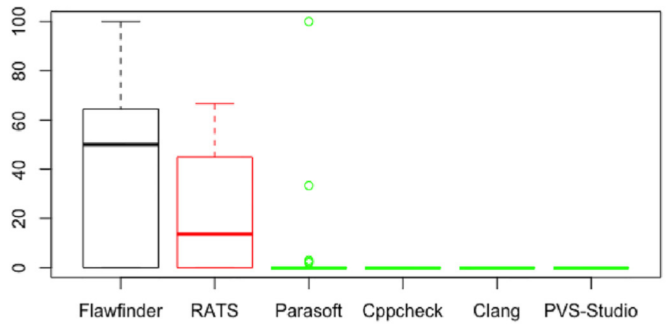
(b) IVR warnings among SAST tools in 2017



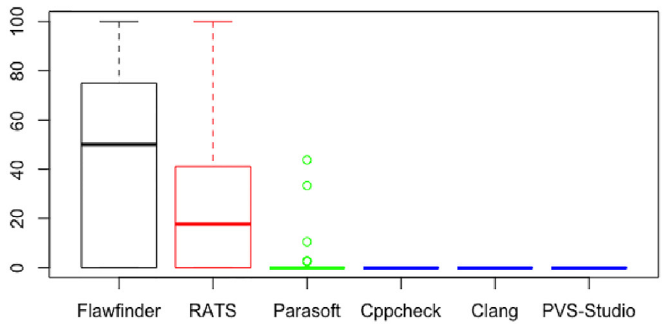
(c) API warnings among SAST tools in 2012



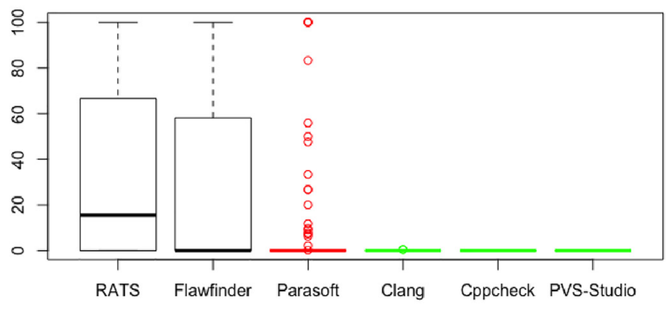
(d) API warnings among SAST tools in 2017



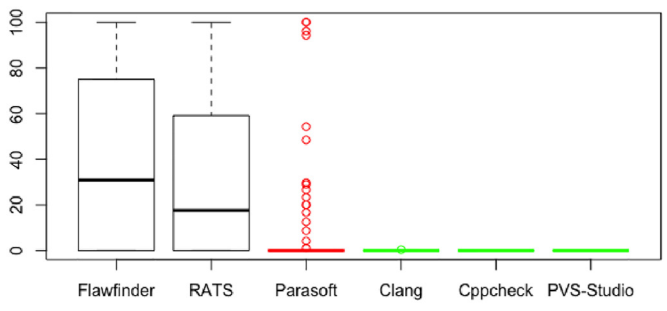
(e) SF warnings among SAST tools in 2012



(f) SF warnings among SAST tools in 2017

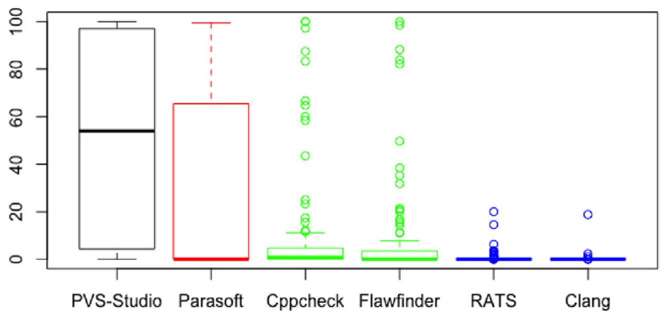


(g) TS warnings among SAST tools in 2012

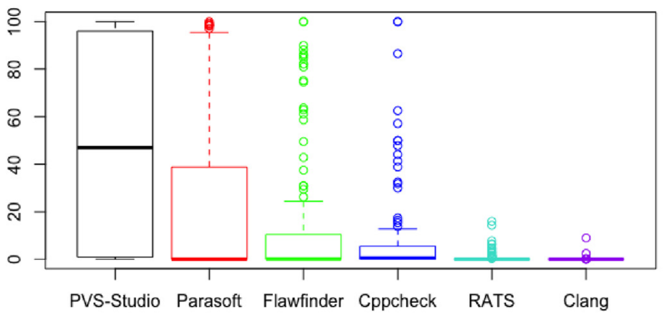


(h) TS warnings among SAST tools in 2017

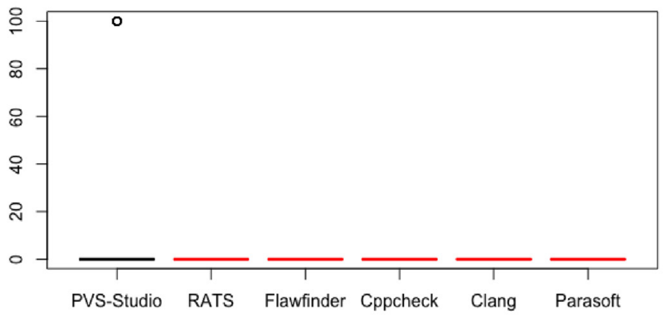
Fig. 14. Box-and-whisker plot and Scott–Knott ESD of the number of bugs classified as Input Validation and Representation (IVR), API Abuse (API), Security Feature (SF), and Time and State (TS). In this figure, the *x*-axis represents normalized number of bug(bugs/LOC), and the *y* represents different SAST tools. Each data point represents a project. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



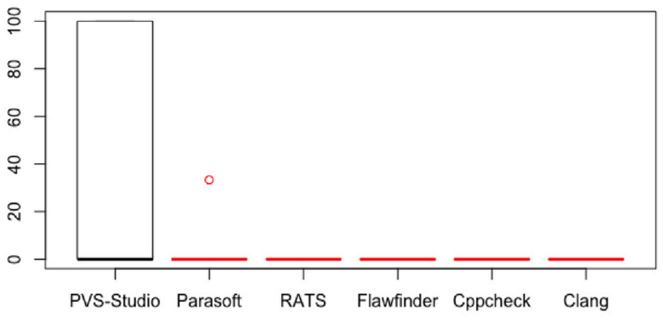
(a) CQ warnings among SAST tools in 2012



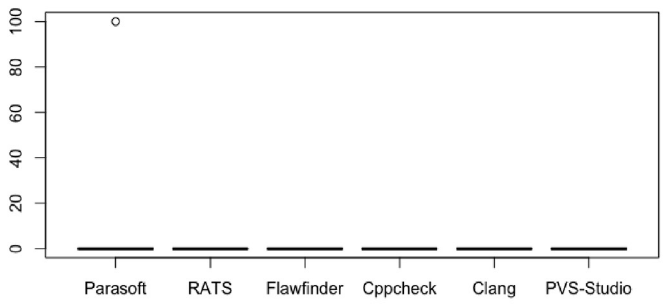
(b) CQ warnings among SAST tools in 2017



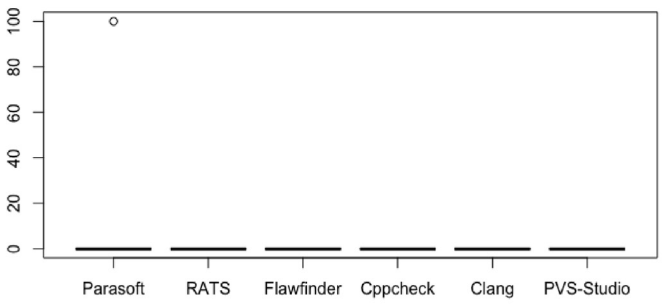
(c) ERR warnings among SAST tools in 2012



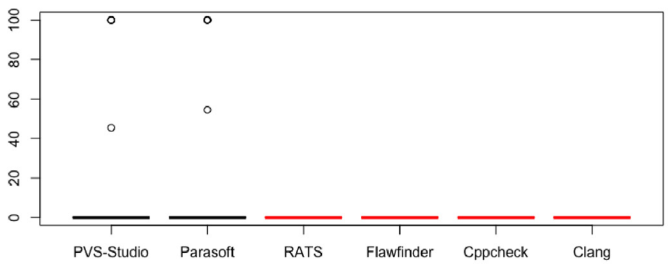
(d) ERR warnings among SAST tools in 2017



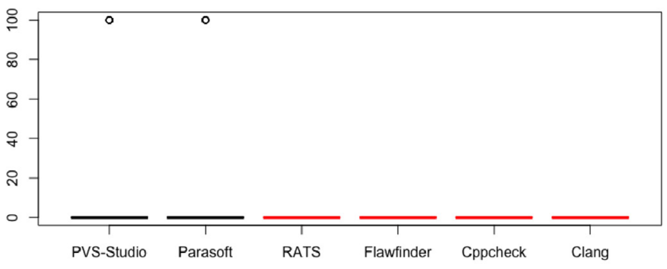
(e) ENC warnings among SAST tools in 2012



(f) ENC warnings among SAST tools in 2017



(g) ENV warnings among SAST tools in 2012



(h) ENV warnings among SAST tools in 2017

Fig. 15. Box-and-whisker plot and Scott–Knott ESD of the number of bugs classified as Code Quality (CQ), Errors (ERR), Encapsulation (ENC), and Environment (ENV). In this figure, the x-axis represents normalized number of bug(bugs/LOC), and the *y* represents different SAST tools. Each data point represents a project. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Appendix C. Bug warning rates among SAST tools (RQ2)

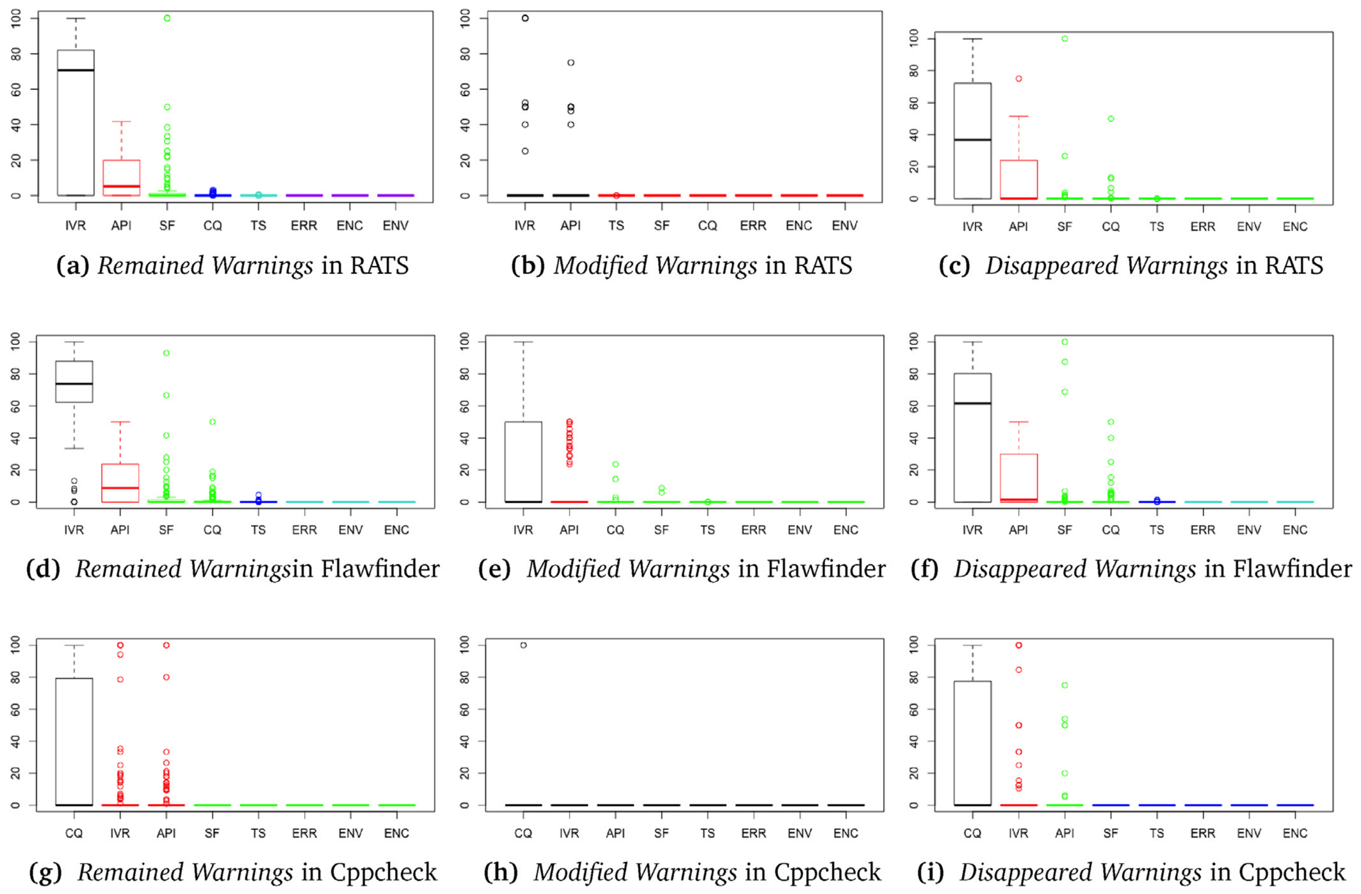
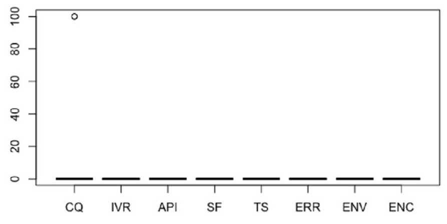
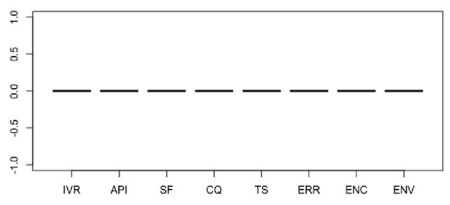


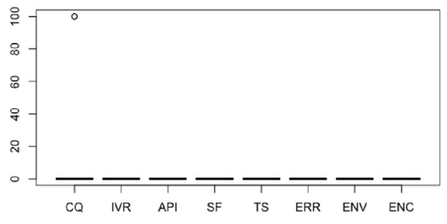
Fig. 16. Box-and-whisker plot and Scott–Knott ESD of the *Remained Warnings* (FP), *Modified Warnings* and *Disappeared Warnings* (TP) among different bug categories by RATS, Flawfinder, Cppcheck. In this figure, the x-axis represents normalized number of bug(bugs/LOC), and the *y* represents different classes of SPK bugs. Each data point represents a project. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



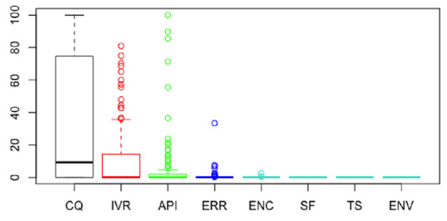
(a) *Remained Warnings*in the Clang Static Analyzer



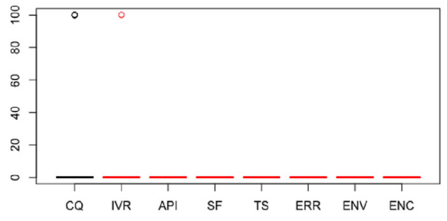
(b) *Modified Warnings* in the Clang Static Analyzer



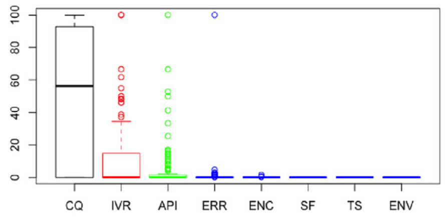
(c) *Disappeared Warnings* in the Clang Static Analyzer



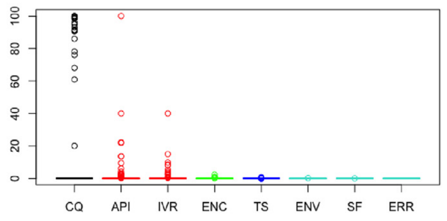
(d) *Remained Warnings* in the PVS-Studio



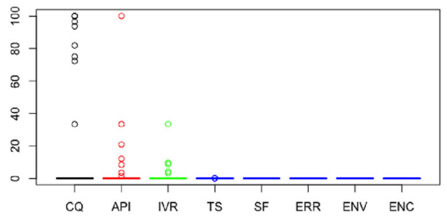
(e) *Modified Warnings* in the PVS-Studio



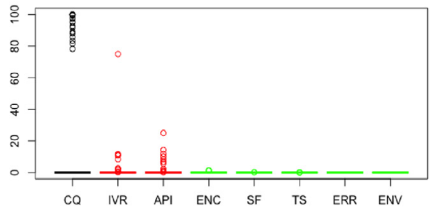
(f) *Disappeared Warnings* in the PVS-Studio



(g) *Remained Warnings* in the Parasoft C/C++test



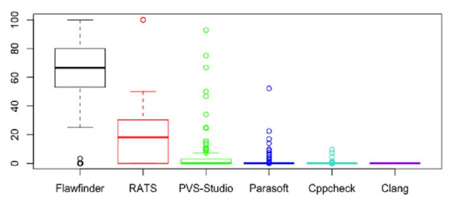
(h) *Modified Warnings* in the Parasoft C/C++test



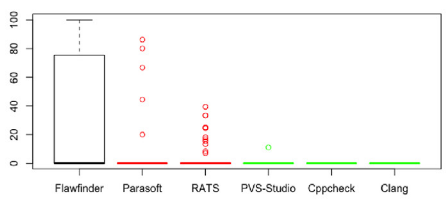
(i) *Disappeared Warnings* in the Parasoft C/C++test

Fig. 17. Box-and-whisker plot and Scott–Knott ESD of the *Remained Warnings* (FP), *Modified Warnings* and *Disappeared Warnings* (TP) among different bug categories by Clang Static Analyzer, PVS-Studio, and Parasoft C/C++ test. In this figure, the x-axis represents normalized number of bug(bugs/LOC), and the *y* represents different classes of SPK bugs. Each data point represents a project. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

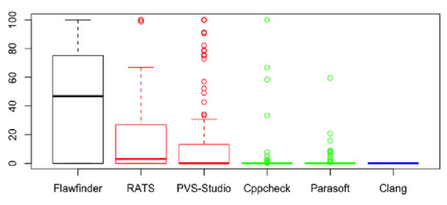
Appendix D. Bug warning rates among SPK categories (RQ2)



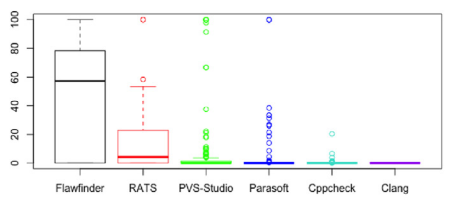
(a) *Remained Warnings* among IVR



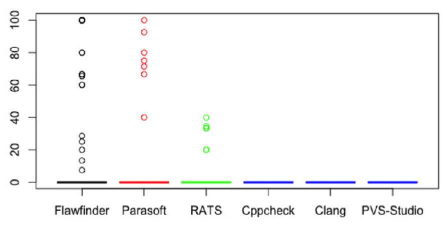
(b) *Modified Warnings* among IVR



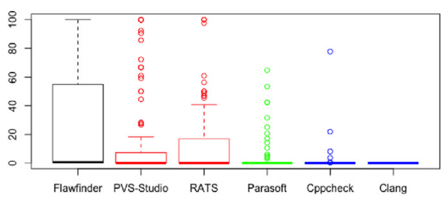
(c) *Disappeared Warnings* among IVR



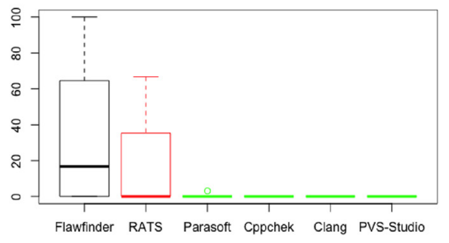
(d) *Remained Warnings* among API



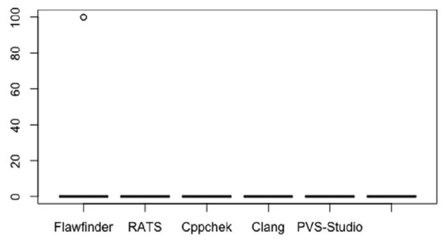
(e) *Modified Warnings* among API



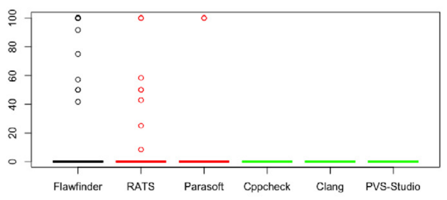
(f) *Disappeared Warnings* among API



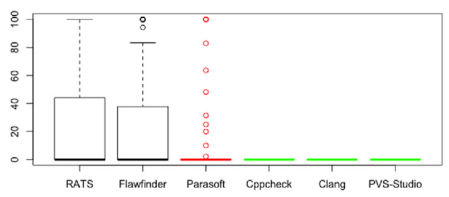
(g) *Remained Warnings* among SF



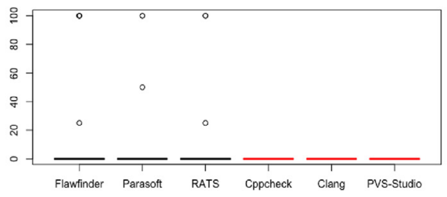
(h) *Modified Warnings* among SF



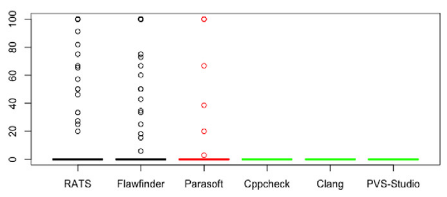
(i) *Disappeared Warnings* among SF



(j) *Remained Warnings* among TS

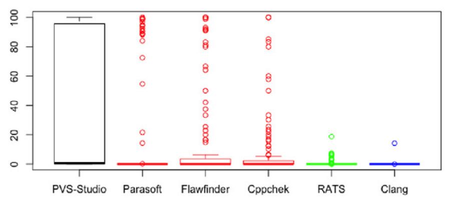


(k) *Modified Warnings* among TS

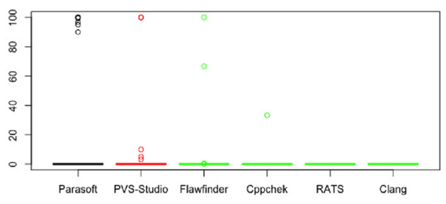


(l) *Disappeared Warnings* among TS

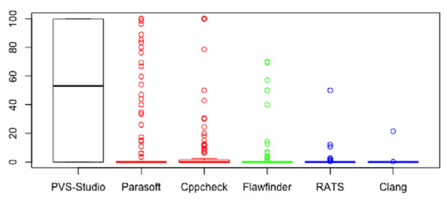
Fig. 18. Box-and-whisker plot and Scott–Knott ESD of the *Remained Warnings* (FP), *Modified Warnings* and *Disappeared Warnings* (TP) among different bug categories among Input Validation and Representation (IVR), API Abuse (API), Security Features (SF), and Time and State (TS). In this figure, the x-axis represents normalized number of bug(bugs/LOC), and the *y* represents different SAST tools. Each data point represents a project. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



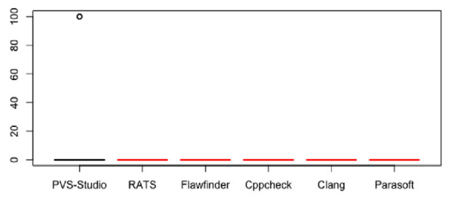
(a) *Remained Warnings* among CQ



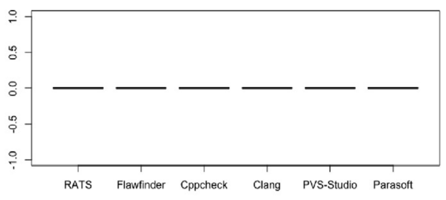
(b) *Modified Warnings* among CQ



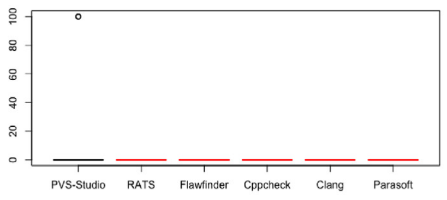
(c) *Disappeared Warnings* among CQ



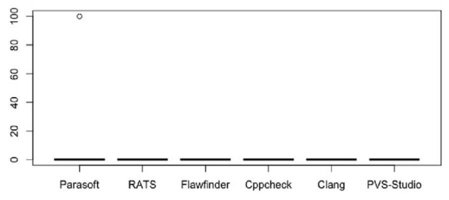
(d) *Remained Warnings* among ERR



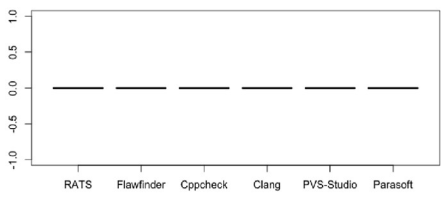
(e) *Modified Warnings* among ERR



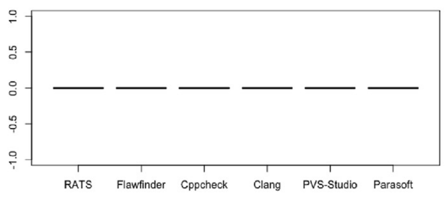
(f) *Disappeared Warnings* among ERR



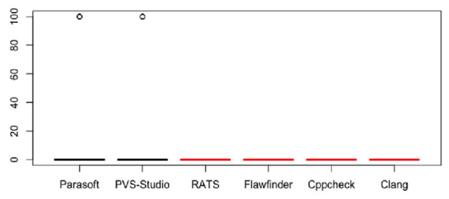
(g) *Remained Warnings* among ENC



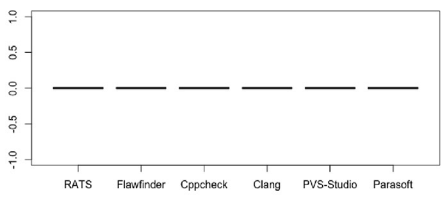
(h) *Modified Warnings* among ENC



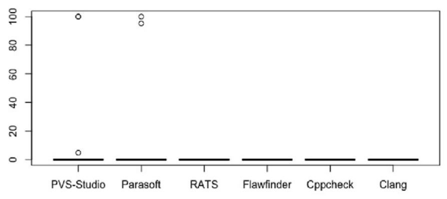
(i) *Disappeared Warnings* among ENC



(j) *Remained Warnings* among ENV



(k) *Modified Warnings* among ENV



(l) *Disappeared Warnings* among ENV

Fig. 19. Box-and-whisker plot and Scott–Knott ESD of tthe *Remained Warnings* (FP), *Modified Warnings* and *Disappeared Warnings* (TP) among different bug categories among Code Quality (CQ), Errors (ERR), Encapsulation (ENC), and Environment (ENV). In this figure, the x-axis represents normalized number of bug(bugs/LOC), and the *y* represents different SAST tools. Each data point represents a project. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

References

Aloraini, B., Nagappan, M., German, D., Hayashi, S., Higo, Y., 2019. An Empirical Study of Security Warnings from Static Analysis tools. https://github.com/BushraAloraini/SAST_Warnings.

Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y., 2007. 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. ACM, pp. 1–8.

Bachmann, A., Bird, C., Rahman, F., Devanbu, P., Bernstein, A., 2010. The missing links: bugs and bug-fix commits. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software engineering. ACM, pp. 97–106.

Baptista, A., 2017. Report: Software Failures Cost $1.1 Trillion in 2016. http://servicevirtualization.com/report-software-failures-cost-1-1-trillion-2016.

Canfora, G., Cerulo, L., Di Penta, M., 2009. Tracking your changes: a language-independent approach. IEEE Softw. 26 (1).

Christakis, M., Bird, C., 2016. What developers want and need from program analysis: an empirical study. In: Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on. IEEE, pp. 332–343.

Clang Static Analyzer. https://clang-analyzer.llvm.org.

Common Weakness Enumeration (CWE). https://cwe.mitre.org.

Coverity Static Analysis. https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html.

Cousot, P., Cousot, R., 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of programming languages. ACM, pp. 238–252.

Di Penta, M., Cerulo, L., Aversano, L., 2009. The life and death of statically detected vulnerabilities: an empirical study. Inf. Softw. Technol. 51 (10), 1469–1484.

Díaz, G., Bermejo, J.R., 2013. Static analysis of source code security: assessment of tools against samate tests. Inf. Softw. Technol. 55 (8), 1462–1476.

Edwards, N., Chen, L., 2012. An historical examination of open source releases and their vulnerabilities. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. ACM, pp. 183–194.

Evans, D., Larochelle, D., 2002. Improving security using extensible lightweight static analysis. IEEE Softw. 19 (1), 42–51.

German, D.M., Adams, B., Stewart, K., 2019. cregit: token-level blame information in GIT version control repositories. Empir. Softw. Eng. doi:10.1007/s10664-019-09704-x.

Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R., 2013. Why don't software developers use static analysis tools to find bugs? In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 672–681.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D., 2014. The promises and perils of mining GitHub. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp. 92–101.

Kildall, G.A., 1973. A unified approach to global program optimization. In: Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM, pp. 194–206.

Kim, S., Ernst, M.D., 2007. Prioritizing warning categories by analyzing software history. In: Proceedings of the Fourth International Workshop on Mining Software Repositories. IEEE Computer Society, p. 27.

Kratkiewicz, K., Lippmann, R., 2005. Using a diagnostic corpus of C programs to evaluate buffer overflow detection by static analysis tools. In: Workshop on the Evaluation of Software Defect Detection Tools, pp. 1–7.

Marjamki, D., Cppcheck. http://cppcheck.sourceforge.net.

Mathworks., Polyspace Bug Finder. https://www.mathworks.com/products/polyspace-bug-finder.html.

Murphy-Hill, E., Zimmermann, T., Bird, C., Nagappan, N., 2015. The design space of bug fixes and how developers navigate it. IEEE Trans. Softw. Eng. 41 (1), 65–81.

Nagappan, M., Munaiah, N., Cabrey, C., Kroh, S., Parikh, N., 2016. Reporeapers. https: //reporeapers.github.io/results/1.html.

National Security Agency Center for Assured Software, 2011. On Analyzing Static Analysis Tools.

Ozment, A., Schechter, S.E., 2006. Milk or wine: does software security improve with age? In: USENIX Security Symposium, pp. 93–104.

Parasoft C/C++ test. https://www.parasoft.com/products/ctest.

PVS-Studio Analyzer. https://www.viva64.com/en/pvs-studio.

Ray, B., Posnett, D., Filkov, V., Devanbu, P., 2014. A large scale study of programming languages and code quality in GitHub. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 155–165.

Rough Auditing Tool for Security (RATS). https://security.web.cern.ch/security/recommendations/en/codetools/rats.shtml.

Soni, M., 2006. Defect Prevention: Reducing Costs and Enhancing Quality, 19. iSixSigma. com

Spacco, J., Hovemeyer, D., Pugh, W., 2006. Tracking defect warnings across versions. In: Proceedings of the 2006 International Workshop on Mining Software Repositories. ACM, pp. 133–136.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2017. An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Softw. Eng. 43 (1), 1–18.

Thornburg, H., 2019. 2019 Global Developer Report: Devsecops.

TIOBE Index for May 2019, programming language C++ is doing well in the TIOBE index. https://www.tiobe.com/tiobe-index.

Torri, L., Fachini, G., Steinfeld, L., Camara, V., Carro, L., Cota, É., Box, P., Alegre, P., 2010. An evaluation of free/open source static analysis tools applied to embedded software. In: Test Workshop (LATW), 2010 11th Latin American, pp. 1–6. doi:10.1109/LATW.2010.5550368.

Tsipenyuk, K., Chess, B., McGraw, G., 2005. Seven pernicious kingdoms: a taxonomy of software security errors. IEEE Secur. Privacy 3 (6), 81–84.

Viega, J., Bloch, J.T., Kohno, T., McGraw, G., 2002. Token-based scanning of source code for security problems. ACM Trans. Inf. Syst. Secur. 5 (3), 238–261. doi:10.1145/545186.545188.

Wagner, D.A., Foster, J.S., Brewer, E.A., Aiken, A., 2000. A first step towards automated detection of buffer overrun vulnerabilities.. In: NDSS, pp. 2000–2002.

Wheeler, D. A., Flawfinder. http://www.dwheeler.com/flawfinder.

Xie, Y., Chou, A., Engler, D., 2003. Archer: using symbolic, path-sensitive analysis to detect memory access errors. ACM SIGSOFT Softw. Eng. Not. 28 (5), 327–336.

Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., Bairavasundaram, L., 2011. How do fixes become bugs? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ACM, pp. 26–36.

Zitser, M., Lippmann, R., Leek, T., 2004. Testing static analysis tools using exploitable buffer overflows from open source code. ACM SIGSOFT Softw. Eng. Not. 29 (6), 97. doi:10.1145/1041685.1029911.