Survey of Approaches for Postprocessing of Static Analysis Alarms

TUKARAM MUSKE, Tata Consultancy Services, India
ALEXANDER SEREBRENIK, Eindhoven University of Technology, The Netherlands

Static analysis tools have showcased their importance and usefulness in automated detection of defects. However, the tools are known to generate a large number of alarms which are warning messages to the user. The large number of alarms and cost incurred by their manual inspection have been identified as two major reasons for underuse of the tools in practice. To address these concerns plentitude of studies propose postprocessing of alarms: processing the alarms after they are generated. These studies differ greatly in their approaches to postprocess alarms. A comprehensive overview of the approaches and techniques to implement them is, however, missing.

In this article, we review 130 primary studies that propose postprocessing of alarms. The studies are collected by combining keywords-based database search and snowballing. We categorize approaches proposed by the collected studies into six main categories: *clustering, ranking, pruning, automated elimination of false positives, combination of static and dynamic analyses,* and *simplification of manual inspection.* We provide overview of the categories and sub-categories identified for them, their merits and shortcomings, and different techniques used to implement the approaches. Furthermore, we provide (1) guidelines for selection of the postprocessing techniques by the users/designers of static analysis tools; and (2) directions that can be explored by the researchers.

CCS Concepts: • **Theory of computation** → **Program analysis**; **Program verification**; **Verification by model checking**; • **Software and its engineering** → **Formal software verification**; **Software verification and validation**; • **Security and privacy** → **Vulnerability scanners**.

Additional Key Words and Phrases: Static analysis, static analysis alarms, postprocessing of alarms, literature search, keywords-based search, snowballing

ACM Reference Format:
Tukaram Muske and Alexander Serebrenik. 2020. Survey of Approaches for Postprocessing of Static Analysis Alarms. 1, 1 (October 2020), 42 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Static analysis tools have showcased their importance and usefulness in automated detection of common programming errors like division by zero and dereference of a null pointer [44, 153]. Furthermore, these tools are also useful to prove absence of errors in safety/security-critical systems [19, 41, 89]. However, these tools are known to be underused in practice [16, 31, 73, 92, 96]. Indeed, these tools generate a large number of alarms warning the user about potential errors [16, 45, 73, 95, 96]. In general there are 40 alarms for every thousand lines of code [16], and 35% to 91% of alarms are false positives [66]. Partitioning alarms into false positives and errors requires manual inspection which is tedious, time-consuming [42, 123, 149, 160], and can be error-prone

Authors' addresses: Tukaram Muske, Tata Consultancy Services, 54B, TRDDC, Hadapsar I.E., Pune, Maharashtra, India; Alexander Serebrenik, Eindhoven University of Technology, Groene Loper 5, 5612 AE, Eindhoven, The Netherlands.

XXXX-XXXX/2020/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[42]. The large number of alarms reported by the tools and cost involved in their manual inspection have been observed to be the major reasons for the underuse of static analysis tools in practice [16, 31, 73, 92, 96].

Improving precision of static analysis tools has been extensively considered in the literature [13, 60, 110]. However, given that verification problems are undecidable in general, reporting of false alarms by these tools is inevitable [42, 149]. Therefore, since last two decades, *postprocessing of alarms*—processing the alarms after they are generated—is being explored as an alternative to address the problem of alarms and the cost associated with their manual inspection. We define postprocessing of alarms as below.

We call processing of alarms after they are generated by a static analysis tool *postprocessing*, if the processing relates to any of the following activities.

(1) *Reducing number of alarms* by before reporting them to users.
(2) *Reducing manual inspection effort* by enriching alarms with additional information.
(3) *Simplifying manual inspection of alarms* by providing assistance and tool support during the inspection process.

Considering the benefits offered by postprocessing of alarms, a plentitude of approaches have been proposed [45, 66], and techniques to implement those approaches differ greatly. However, to the best of our knowledge, a comprehensive overview of these approaches and techniques to implement them is missing. Indeed, existing surveys of static analysis techniques either focus on specific postprocessing techniques [3, 45, 66] or discuss static analysis in general [39, 104]. The only notable exception is our previous work [128] the current article builds on and extends. In the absence of such an overview, (1) developers and users of static analysis tools have a hard time choosing postprocessing techniques from the plentitude of existing ones, and (2) researchers might be rediscovering existing approaches or miss opportunities to explore new directions.

Therefore, in this article we ask the following research question.

RQ: What approaches have been proposed for postprocessing of alarms?

To understand the current state of alarms postprocessing, we performed a *systematic literature search* combining *keywords-based database search* [86, 113] and *snowballing* [11, 176]. We combine the approaches to complement their strengths: the results of the former provided a start set required in the latter, and the latter identified the relevant papers which were missed by the former. The literature search was performed initially during the period of June 4 to June 14, 2016. We extended the literature search to include the relevant studies that have been published after the initial search. The extended search is performed during the period of Dec 24, 2019 to Jan 10, 2020. We call the first search *initial literature search* and the extended one *extended literature search*.

Through the two literature searches, we identify 130 primary studies (research papers) that propose postprocessing of alarms, and identify six main categories of approaches: *clustering, ranking, pruning, automated elimination of false positives, combination of static and dynamic analyses,* and *simplification of manual inspection*. Furthermore, we categorize five of those categories into two or more sub-categories depending on techniques used to implement the approaches. We provide an overview of the categories and sub-categories, their merits and shortcomings, and different techniques used in their implementations. We observe that the approaches identified are complementary and can be combined in different ways. We provide (1) guidelines for selection of the postprocessing techniques by the users and designers of static analysis tools; and (2) directions that can be explored by the researchers.

The work in this article builds on our prior survey published in the proceedings of SCAM 2016 [128]. The prior survey has been conducted with a larger scope of *handling of alarms*. The survey is based on the initial literature search that identified 79 primary studies as relevant. In Section 2 we elaborate on the differences between the current work and the prior survey [128]. The work in this article has also been included as a chapter in PhD dissertation of the first author [122].

The contributions of this article are threefold.

(1) The primary contribution of this work is the categorization of 130 research papers proposing approaches for postprocessing of alarms.
(2) We identify six main categories of approaches: clustering, ranking, pruning, automated elimination of false positives, combination of static and dynamic analyses, and simplification of manual inspection.
(3) We observe that the approaches to postprocess alarms are complementary and can be combined in different ways for better results.

Article Outline. Section 2 discusses related work highlighting differences between our survey and other similar surveys in the area. Section 3 describes the methodology used to conduct the initial and extended literature searches. Section 4 provides overview of data extracted from the relevant studies collected from the two literature searches. Section 5 describes the identified categories of approaches for postprocessing of alarms. Section 6 summarizes merits and shortcomings of the approaches, presents guidelines for selection of the approaches, and provides directions for future work. Section 7 concludes.

2 RELATED WORK

In this section, we compare our work with recently published (1) literature reviews of techniques for postprocessing of alarms, and (2) studies about evaluation or benchmarking of tools/techniques that postprocess alarms. We start by comparing this work with our prior survey published in the proceedings of SCAM 2016 [128]. The current survey is performed by limiting the scope to *postprocessing of alarms*, whereas the prior survey was conducted with a larger scope of *handling of alarms*, that included *design of light-weight static analysis tools* also as an approach to handle alarms. We do not consider this approach relevant to postprocessing of alarms. The prior survey is based on the initial literature search that identified 79 primary studies as relevant. In the current survey, wherever appropriate, based on the papers additionally identified as relevant during the extended literature search, we have also updated the sub-categories of those six main categories. Thus, the (sub)-categorization presented in this article is different from the (sub)-categorization in the prior work [128] and also is as per our improved understanding of postprocessing of alarms (Section 3.1.3).

The systematic literature review conducted by Heckman and Williams [66] reviews techniques for ranking and pruning of alarms. In this review, approaches proposed by 21 different studies for postprocessing of alarms, are analyzed and categorized into two categories, ranking and pruning. Among those 21 studies, 10 studies propose classification of alarms into actionable or non-actionable classes, while the other 11 studies propose ranking of alarms. Compared to this review, our literature survey is more comprehensive as it includes more studies (130) that propose a variety of approaches for alarms postprocessing. For example, due to inclusion of those additional studies, we could identify new categories like clustering, automated false positives elimination, and simplification of manual inspection. Moreover, wherever suited, the studies in each category are further categorized into multiple sub-categories. This categorization helps to understand the proposed approaches better and comprehensively. Five of those 21 studies included in the review by Heckman and Williams [66] were not included in our study: the excluded studies performed evaluations of static analysis tools (e.g. [64] and [135]) rather than introducing new postprocessing techniques.

In their mapping study, Mendonca et al. [39] have selected and analyzed 51 studies to identify state-of-the-art static analysis techniques and tools, and main approaches developed for postprocessing of alarms. In our study, as our focus was on different approaches through which alarms are postprocessed, we did not include 39 of those 51 studies. The excluded studies dealt with improving analysis precision, study of defects, and even evaluation and benchmarking of static analysis tools. In their mapping study, Elberzhager et al. [45] have classified and provided analysis of approaches that combine static analysis and dynamic quality assurance techniques. The static quality assurance techniques deal with code reviews, inspections, walkthroughs, and usage of static analysis tools, whereas our literature survey is with much broader scope of postprocessing of alarms: in our survey, combination of static and dynamic analyses is one of the categories of approaches to postprocess alarms.

Li et al. [104] have performed a systematic literature review to provide overview of state-of-the-art works that statically analyze Android apps. From these works, they highlight the trends of static analysis approaches, pinpoint where the focus has been put, and enumerate the key aspects where future research is still needed. In this review, 124 research papers are studied. The review is performed with much broader scope: the review is performed mainly in the following five dimensions (1) problems targeted by the approach, (2) fundamental techniques used by authors, (3) static analysis sensitivities considered, (4) Android characteristics taken into account, and (5) the scale of evaluation performed. Unlike to this review, our study is focused on understanding the state of postprocessing of alarms irrespective of domains of the applications being analyzed.

To the best our knowledge, there does not exist other literature survey or reviews studying alarms postprocessing approaches. Several other studies, like [3, 106, 182] evaluate various techniques for alarms postprocessing. Allier et al. [3] have proposed a framework to compare different alarms ranking techniques and identify the best approach for ranking alarms. The various techniques proposed for postprocessing of alarms are compared using a benchmark having programs in Java and Smalltalk, and three static analysis tools: FindBugs, PMD, and SmallLint. Using this framework, algorithms to rank alarms are compared. In another study, Liang et al. [106] have proposed an approach to construct a training set automatically, required for effectively computing the learning weights for different impact factors. These weights are used later to compute scores used in ranking/pruning of alarms. As opposed to these studies, our literature survey studies approaches that have been proposed for postprocessing of alarms.

As compared to the existing reviews and studies which aimed at understanding postprocessing of alarms, our presented study is based on more papers and describes multi-level categorization of the approaches. Therefore, it can help the users and designers/developers of static analysis tools to choose the postprocessing approaches suited in their work.

3 METHODOLOGY

This section discusses the methodolgoy we used to (1) conduct the literature search and collect studies that propose techniques for postprocessing of alarms, and (2) extract data from the relevant studies.

The literature search is performed by combining *keywords-based database search* [86, 113] and *snowballing* [11, 176]. Performing a literature search is a time consuming activity. Moreover, it requires the searcher to be an expert in the area (postprocessing of alarms) and finding such a searcher is a hard task. Therefore, the literature search is performed by the first author without involving additional experts. The first author has 10 years of experience in (1) research in designing and developing new techniques for postprocessing of alarms [121, 123, 125–127, 129, 130], and (2) developing a commercial static analysis tool (TCS ECA [161]).

Table 1. Keywords used during the keywords-based database search.

Category	Main keywords identified from the research question	Relevant keywords to be used literature search (Search keywords)
I	postprocessing goals	1) elimination, 2) reduction, 3) simplification, 4) ranking, 5) classification, 6) reviewing, 7) inspection
II	static analysis	1) static analysis, 2) automated code analysis, 3) source code analysis, 4) automated defects detection
III	alarms	1) alarm, 2) warning, 3) alert

Henceforth in the article, we use *studies* and *(research) papers* interchangeably. In postprocessing of alarms, we consider alarms generated by both *code proving tools* (i.e., tools such as Astrée [33] and Polyspace Code Prover [164] that can be used for proving absence of bugs of certain types), and *bug finding tools* (i.e., tools such as FindBugs [9] and Lint [74] that are used for finding bugs).

3.1 The Initial Literature Search

The initial literature search was conducted during the period of June 4, 2016 to June 14, 2016.

3.1.1 Keywords-based Database Search. Inspired by systematic literature reviews [86, 113], we conducted a keywords-based database search in Google Scholar[1] to collect the papers that propose techniques for postprocessing of alarms. We call these papers *relevant papers*. The keywords that we used during the search are listed in Table 1, and are identified from the research question (Section 1).

The keywords identification is discussed next. Both the authors of the article working together identified three main keywords from the research question (RQ): postprocessing, static analysis, and alarm. These main keywords are shown in three categories I, II, and III respectively. Corresponding to each of the main keywords, we then identified search keywords: the keywords to be used during the keywords-based database search.

The search keywords in Category I, related to *postprocessing goals*, are identified referring to the definition of *postprocessing of alarms* presented in Section 1. We identified *reduction* based on Cases (1) and (2) in the definition, and *elimination* as its synonym. Keyword *inspection* is identified based on Cases (2) and (3), and *reviewing* as its synonym. Keyword *simplification* is identified based on Case (3). In addition, we consulted the recently published literature reviews and studies (discussed in Section 2), and identified *ranking* and *classification*, mainly from the related work [66] which identifies ranking and classification as two alarms postprocessing approaches.

The search keywords in Categories II and III are identified as synonyms to the main keywords *static analysis* and *alarms* respectively.

The selection of keywords, shown in Table 1, results in $84 = 7 \times 4 \times 3$ different search strings. We manually searched each of the search strings in Google Scholar, and examined the first 150 results of every search. To avoid examining same paper multiple times, that appears in results of multiple search strings, we enabled highlighting of the visited links in the browser. We ignored the links (search results) that were marked as visited. Thus, during this process, we examined a total of 12600 results including the duplicates.[2]

For each paper in the results that were not identified as duplicates, we checked whether the paper should be included in the collection of *relevant papers*. We identified a paper as *relevant* only if

[1] Google Scholar. https://scholar.google.com/

[2] Since the keywords-based search is performed manually, we did not explicitly measure the number of duplicates in the search results.

- it proposes a technique, method, or an approach to postprocess alarms; and
- it is a peer-reviewed paper.

We excluded a paper from the collection of relevant papers if it deals with

- improving precision of the underlying static analyses like value analysis and pointer analysis, or refinements to the analyses (like [60, 77, 110]);
- an approach or methodology followed to reduce the number of alarms by designing light-weight static analysis tools;
- fault prediction or error/bug report triaging;
- mining of bug repositories in the context of software maintenance/evolution;
- study of economics or benefits of usage of static analysis tools (like [92, 192]); or
- evaluation, comparison, or benchmarking of precision of various static analysis tools (such as [24, 151, 162]).

While applying the inclusion/exclusion criteria for each paper, we considered the title, abstract, introduction/motivation, conclusion, and sometimes evaluation section of the paper. In the case of a paper satisfying both the inclusion and exclusion criteria, the paper was deemed to be relevant. This keywords-based search led to identification of 46 relevant papers.

3.1.2 Snowballing. After the keywords-based search, we performed snowballing [11, 176] due to the following reasons: (a) the search strings considered based on the keywords in Table 1, might be incomplete, e.g., due to terminological differences among the papers; and (b) more importantly, given a good *start set*, snowballing approach is found to be more effective and efficient in collecting relevant papers as compared to the keywords-based searches [11, 176]. By conducting snowballing after the keywords-based database search, we tried to identify and include as many relevant papers as possible, which were missed by the database search [94].

Creation of Start Set. To begin with, a literature search using snowballing requires a start set having diversity in the included papers to avoid bias towards any specific class of papers and thus bias towards any specific approaches identified from them. Moreover, such a start set reduces the risk of missing a paper from clusters of papers not referring to each other [176]. In our literature search, we created the required start set by including all the relevant papers identified through the earlier keywords-based search. Thus, the start set used to perform snowballing included 46 relevant papers.

Backward and Forward Snowballing. After the start set is created, we performed iterations of forward and backward snowballing. In the backward snowballing, the papers in *the reference list* of each relevant paper are examined to identify new papers to be included. In the forward snowballing, papers citing an included paper are examined to identify new relevant papers (citations analysis). We performed the citations analysis using Google Scholar. During the snowballing, we used the same inclusion/exclusion criteria that were used during the earlier keywords-based search.

In snowballing, iterations of the backward and forward snowballing are performed till saturation has been reached, i.e., next iteration of snowballing is not be performed when current iteration does not identify a new relevant paper. In the snowballing we conducted, two iterations of the backward and forward snowballing were sufficient: the second iteration of the forward and backward snowballing did not identify any new paper as relevant. During the snowballing process ca. 5800 papers[3] were examined for inclusion. With this search we identified 26 new relevant papers. This activity demonstrates that the combination of the two search approaches helped each approach to complement the other.

[3] The number includes duplicates in the search results.

Table 2. Summary of results of the literature searches conducted.

Literature searches conducted		Number of papers examined including duplicates	Number of new identified relevant papers
Initial literature search	Keywords-based search	12600	46
	Snowballing	5800	26
Extended literature search	Keywords-based search	12600	35
	Snowballing	2345	23
		Total	**130**

3.1.3 Relevant Papers Identified. Therefore, based on our initial literature search, performed by combining the two search approaches, we identified 72 papers that propose techniques for postprocessing of alarms.

In fact, the initial search has been published in the proceedings of SCAM 2016 [128]. This search was performed with a larger scope of *handling of alarms*, which included *designing of light-weight static analysis tools* also as an approach to handle alarms. This search identified 79 relevant papers. The number of papers relevant to *postprocessing of alarms*, 72, is identified after (1) excluding five papers that belonged to the additional approach (design of light-weight static analysis tools), and (2) excluding two relevant papers based on our improved understanding of postprocessing of alarms.

3.2 The Extended Literature Search

We performed the extended literature search to include relevant papers that are published after the initial literature search, i.e., after June 14, 2016. This extended search was performed between Dec 24, 2019 and Jan 10, 2020. To conduct the extended search, we followed the same methodology used to conduct the initial literature search.

3.2.1 Keywords-based Database Search. Using the same keywords in Table 1, we performed keywords-based search at Google Scholar. For each of the 84 search strings, we examined the papers that are published after the initial literature search, because the papers published before 2016 are already examined in the initial literature search. We used Google Scholar's filter option (Since 2016) to include in results only those papers that are published in 2016 or onwards. From the search results (i.e., papers that are published in 2016 or onwards), we considered and examined the first 150 results to identify relevant papers. During identification of the relevant papers, we used the same inclusion and exclusion criteria (Section 3.1.1). This search led to identification of 35 new relevant papers.

3.2.2 Snowballing. We performed snowballing using 107 relevant papers as the required start set: 72 papers identified by the initial literature search, and 35 papers identified by the keywords-based search of the extended search. During the forward snowballing we considered only the papers that are published in 2016 or onwards using the Google Scholar's filter option. During this search, two iterations of the forward and backward snowballing got performed, in which we examined 2345 papers[4]. This search led to identification of 23 additional relevant papers.

3.2.3 Relevant Papers Identified. The two literature searches, the initial and extended searches, together led to identification of 130 relevant papers.

Table 2 summarises the results of the literature searches conducted.

[4] The number includes duplicates in the search results.

3.3 Data Extraction

This section describes extraction of data from those 130 relevant papers identified using the literature search (Section 3).

We reviewed each of the relevant papers and extracted the following data: (1) the approach(es) proposed in the paper for postprocessing of alarms, (2) techniques and artifacts used to implement those approaches, (3) static analysis tools used for evaluating the approaches and techniques, and (4) programming languages supported by the tools.

We used open tagging [159] to categorize approaches proposed by the papers. The tagging was performed by the first author. The papers having similar approaches are grouped together, and a broader level approach is identified describing the group. When a paper is found to propose multiple approaches, i.e. the paper can belong to multiple categories, the most prominent approach mostly suggested by the title of the paper is selected to determine the category. For example, for the study by Kremenek et al. [90] that presents clustering and ranking of alarms and utilizes user-feedback for ranking purposes, we have identified ranking as its primary approach. Moreover, we categorized the identified categories further into sub-categories depending on the main characteristics of the approaches or techniques used to implement the approaches.

We used open tagging [159] to categorize approaches proposed by the papers. The tagging was performed by the first author. The papers having similar approaches are grouped together, and a broader level approach is identified describing the group. When a paper is found to propose multiple approaches, i.e. the paper can belong to multiple categories, the most prominent approach mostly suggested by the title of the paper is selected to determine the category. For example, for the study by Kremenek et al. [90] that presents clustering and ranking of alarms and utilizes user-feedback for ranking purposes, we have identified ranking as its primary approach. Moreover, we categorized the identified categories further into sub-categories depending on the main characteristics of the approaches or techniques used to implement the approaches.

4 OVERVIEW OF THE EXTRACTED DATA

This section provides overview of the data extracted from the relevant papers.

As a result of the above categorization (discussed Section 3.3), the following six categories of approaches are identified.

A. *Clustering: Alarms are clustered* into several groups based on similarity or correlations among them.

B. *Ranking: Alarms are ranked* using various characteristics of the alarms, the source code, history of bug/alarm fixes, code-commit history, and so on.

C. *Pruning:* Alarms are classified into two classes, actionable and non-actionable and *the non-actionable alarms are pruned*.

D. *Automated false positives elimination (AFPE):* Alarms are processed further using more precise techniques like model checking and symbolic execution *to automatically identify and eliminate false positives from the alarms*.

E. *Combination of static and dynamic analyses: Alarms are processed using dynamic analysis* to generate test cases that validate true errors.

F. *Simplification of manual inspection: Manual inspection of alarms is simplified* by enriching alarms with additional information, providing tool support, and so on.

Figure 1 presents summary of the categorization of approaches for postprocessing of alarms, along with the number of papers in each category. Moreover, it presents the identified sub-categories and relevant papers belonging to them. In the figure, the sub-categories of approaches identified from relevant papers obtained through the initial (resp. extended) literature search are shown in non-italics

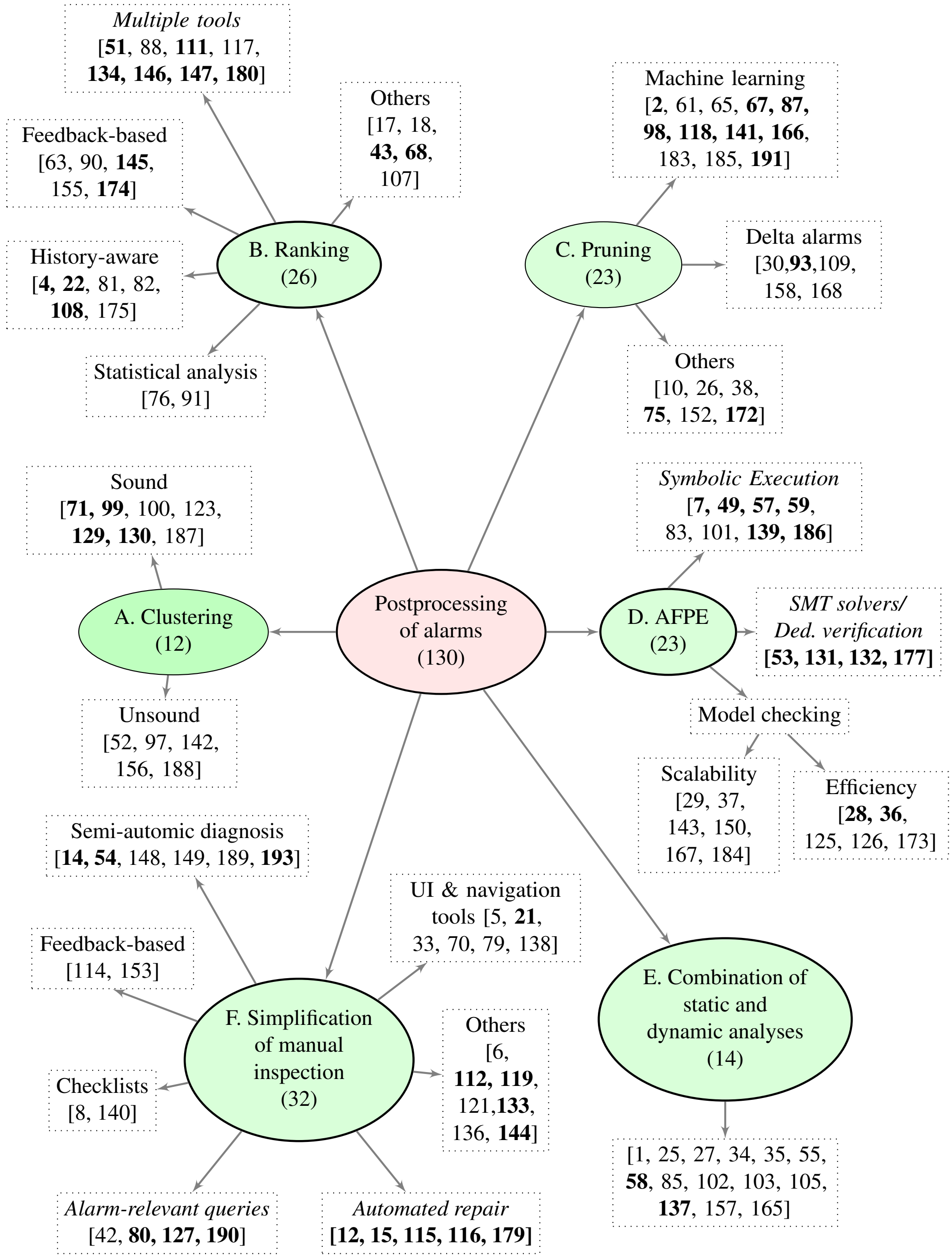

Fig. 1. Summary of the (sub)-categories of approaches proposed for postprocessing of alarms and the corresponding relevant papers.

(resp. *italics*). The relevant papers collected through the extended literature search are shown by marking them in bold. The presented categorization is described in detail in the next section (Section 5).

Figure 2 presents year-wise distribution of the relevant papers per category of the approaches. It indicates that, there is continuous ongoing interest in the topic (postprocessing of alarms), and comparatively a higher number of papers are published recently (in the last three years). Moreover, simplification of manual inspection has been the more popular category comparatively, while ranking, pruning, and AFPE have received nearly equal popularity.

In Table 4 we summarize the following data extracted from the relevant papers.

(1) The relevant papers (Column *Relevant paper (Other sub-categories)*).
(2) The main (sub)-category of the approach identified for the paper (column *Categories*). In case a paper proposes multiple alarms postprocessing approaches belonging to two or more sub-categories, the sub-categories other than the main sub-category identified for the paper are presented in column *Relevant paper (Other sub-categories).*
(3) The year of its publication (column *Year*).
(4) Static analysis tools used to evaluate the proposed approaches, i.e., static analysis tools used to generate alarms required in evaluation of the proposed approaches (column *Tools used in evaluation*).
(5) Programming languages supported by the tools used in evaluation of the approaches (column *Lang.*), i.e., the programming languages of programs on which the alarms used in the evaluation are generated.
(6) Techniques used to implement the proposed approaches (column *Techniques*). Note that the techniques in this column do not correspond to the techniques implemented in static analysis tools (listed in Column *Tools used in evaluation*) that are used to generate alarms used in evaluation of the proposed approaches.
(7) Artifacts used to implement the techniques listed in column *Techniques* (column *Artifacts used*).

In the column *Srch* of Table 4, we denote the search method in which the corresponding paper is identified as relevant. We use I_K and I_S respectively to denote the keywords-based search and snowballing of the initial literature search, and E_K and E_S respectively to denote the keywords-based search and snowballing of the extended literature search.

For example, the first row in Table 4 should be read as follows: The paper by Jiao et al. [71] proposes sound clustering of alarms. The proposed postprocessing approach uses defect models (artifacts) generated using feature functions. The approach is evaluated using alarms generated by DTSC on C programs. The paper has been published in 2017 and included during the extended literature search.

Following are a few observations made from the data summarized in Table 4.

- Out of 130 research papers, 15 research papers propose mulitple alarms postprocessing approaches belonging to two or more sub-categories.
- Static analysis tools that analyze *C* programs are used in 57% of the relevant studies for evaluating the postprocessing approaches and techniques proposed, i.e., *C* is the most popular programming language targeted by alarms postprocessing approaches. In total 65 different tools supporting analysis of *C* programs have been used.
- Java has been the second most popular language during evaluations in 36% of the relevant studies. In total these studies used 34 different static analysis tools to analyze Java programs. Among these tools, FindBugs [9] has been found to be the most popular tool. PMD [4] and

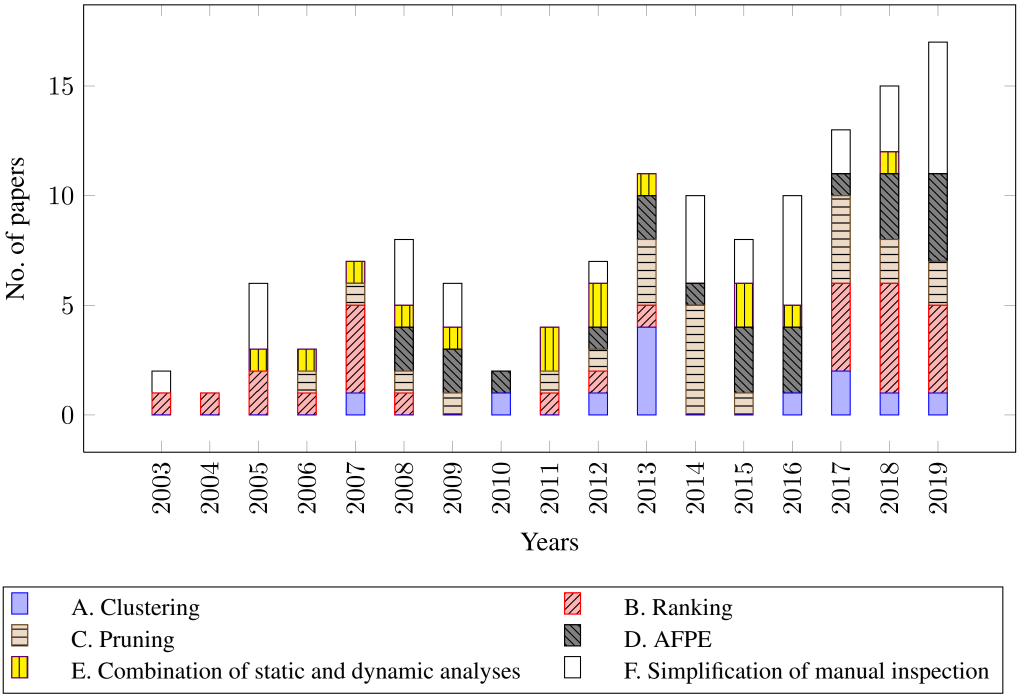

Fig. 2. Number of the relevant papers published year- and category-wise.

Jlint [72] have been other two commonly used tools for Java programs. This indicates that usage of light-weight/shallow static analysis tools is common in analysis of Java programs.

5 DETAILS OF APPROACHES FOR POSTPROCESSING OF ALARMS

In this section, we describe the categories of approaches and their sub-categories (Figure 1), that we identified based on the papers collected through the literature search. For each sub-category, we briefly describe a few relevant papers as its representatives.

5.1 Clustering of Alarms

In this category of the approaches, alarms are clustered into several groups based on similarity or correlation among them. Since alarms in a group are similar/correlated, generally only a few of them need to be inspected [99, 100, 123, 187], or all of them get inspected together [52, 97]. In both cases, clustering of alarms allows to reduce the overall inspection effort.

We categorize the approaches further into *sound* and *unsound*. The categorization depends on whether the clustering approaches guarantee the following relationship among the alarms grouped together: when one or more alarms in a cluster, identified as representative alarms of the cluster, are false positives, all the other alarms in the same cluster are also false positives. We call approaches that guarantee this relationship *sound*, otherwise *unsound*. In other words, we identify a clustering approach as *sound* if the approach, in addition to grouping alarms, also selects a fewer alarms in each group as representatives of the group, guaranteeing that when the representative alarms are false positives, the other alarms in the same group are also false positives. We stress that the notion of soundness/unsoundness of clustering that we introduce for sub-categorization of clustering approaches is different from the soundness/unsoundness of the static analysis that ultimately generated the alarms being grouped.

5.1.1 Sound Clustering.

The approaches in this sub-category cluster alarms such that when one or more representative alarms of a cluster are false positives, all the other alarms in the same cluster are also false positives [99, 100, 123, 129, 130, 187]. The representative alarms of a cluster are called *dominant alarms* of the cluster. The implementation of these approaches is based on analysis techniques like data flow analysis [78] and abstract interpretation [32]. Due to the relationship between dominant and the other alarms in each cluster, inspection of the other alarms is not required when the dominant alarms are found to be false positives. Since there are no false negatives arising due to skipping inspection of alarms other than the dominant alarms, this clustering approach is suitable for both the types of static analysis tools, i.e., code proving tools as well as bug finding tools.

Merits and Shortcomings. In sound clustering, mostly dominant alarms need to be inspected while skipping inspection of the other alarms. The techniques used to implement sound clustering of alarms are generally efficient, and sound clustering techniques are suitable tools to reduce the number of alarms reported by static analysis tools. Reduction in the number of alarms by sound clustering depends on the percentage of alarms identified as dominant.

5.1.2 Unsound Clustering.

This sub-category relates to clustering alarms using similarity in syntactic or structural information, that is produced by static analysis tools or computed separately. This information relates to the code, alarm, or both. Unlike sound clustering, there are no guarantees on the relationship among alarms belonging to the same group. Due to this, skipping inspection of an alarm in a group can result in a false negative.

The techniques implementing this approach use heuristics to group similar alarms together, and propose to inspect those grouped alarms together. For example, Fry et al. [52] have used both structural and syntactic information to partition alarms into groups of related/similar alarms. The partitioning is based on the hypothesis that alarms on the same or similar execution paths may be related and can be inspected together to reduce inspection time. On similar lines, Podelski et al. [142] have proposed a semantics-based signature for an alarm and the signatures are used to group the alarms.

Le and Soffa [97] have used *cause relationships* among the alarms—occurrence of one alarm can cause another alarm to occur—to group the alarms. They first construct a correlation graph by determining the error states of alarms and propagating the effects of the error states along the paths (cause relationships). Then they use the correlation graph to reduce the number of alarms that need to be inspected along a path. However, reducing the number of alarms this way may result in false negatives, because at least one of the faults involved in correlation can be a false positive, and after manual inspection of this false positive, skipping manual inspection of other correlated faults can miss detecting a real fault.

Merits and Shortcomings. Unsound clustering of alarms, performed based on the alarms' similarity in syntactic or structural information, can help to reduce the inspection effort when they are inspected group-wise. However, skipping inspection of some of the alarms in a cluster may result in false negatives.

5.2 Ranking of Alarms

This category corresponds to prioritizing alarms such that the alarms that are more likely to be true errors are ordered up in the list. This approach is of help when not all alarms can be inspected and the inspection time should be spent in an effective way.

5.2.1 Statistical Analysis-based Ranking.

The approaches in this sub-category are based on statistical analysis to rank alarms. For example, Kremenek and Engler [91] have employed a simple statistical model to rank alarms. It is based on the observation that, code containing many successful checks (safe cases analyzed by the tool) and a small number of alarms, tends to contain a real error. As another example, Jung et al. [76] have used a statistical method (Bayesian statistics) to compute probability of an alarm being true, and the probabilities are then used to rank the alarms.

Merits and Shortcomings. The approaches in this sub-category are effective to rank alarms. However, as a shortcoming of the ranking approaches, all the reported alarms need to be manually inspected by the user.

5.2.2 History-aware Ranking.

The approaches in this sub-category use history of alarm fixes as a basis to rank the alarms. For example, Kim and Ernst [81, 82] have ranked alarms by analyzing the software change history, where the categories of alarms that are quickly fixed by the programmers are treated as being more important. On similar lines, Aman et al. [4] estimate lifetimes of alarms by using survival analysis method, and assign higher priority to alarms which have shorter lifetimes: a shorter-life alarm is considered to be more important since many programmers resolved it sooner. Williams and Hollingsworth [175] have proposed a ranking scheme based on commonly fixed bugs and information automatically mined from the source code repository.

Merits and Shortcomings. The approaches in this sub-category allow to identify alarms that are more important to the user, and thus rank them with higher priority. However, implementing them requires history of fixing of alarms and the importance given by the user to the alarms. Identifying such fixes from code commit history can be difficult as commit can have a few other code changes that are not related to alarm fixes. Therefore, identifying such alarm fixes history can be difficult. Also, these approaches may not be applicable to initial versions of the code as fix history is not available.

5.2.3 User Feedback-based Self-adaptive Ranking.

In this sub-category, user feedback is used to rank alarms. For example, Shen et al. [155] have first assigned a predefined defect likelihood for each alarm pattern, and then ranked the alarms based on the defect likelihood. Later, the initial ranking is self-adaptively optimized based on feedback from users. On similar lines, Kremenek et al. [90] have used user-feedback to dynamically reorder ranked alarms after inspection of each alarm. In their technique, Raghothaman et al. [145] first associate each alarm with a confidence value by performing Bayesian inference on a probabilistic model derived from the analysis rules. Later in subsequent iterations, user feedback is captured during inspection of the alarms with the highest confidence, and the feedback is used to recompute the confidences of the remaining alarms. As another example, Heckman [63] has utilized user feedback from analyzed alarms, by combining it with alarm types and code locality, to rank the remaining alarms.

Merits and Shortcomings. The approaches in this sub-category allow to rank alarms based on user feedback. Therefore, user involvement is a must for this postprocessing approach. Also, effectiveness of these approaches mostly depends on how effectively the alarms are ranked initially (by assinging predefined defect likelihood).

5.2.4 Multiple Tools Results-based Ranking.

In this sub-category, results of multiple tools employing different static analysis methods are merged and ranked. In these studies, the merging of results enables results of different tools to validate each other, which in turn, greatly increases or decreases confidence about false positives and false negatives [51, 88, 134, 180]. In this sub-category we also include the studies that merge results of multiple static analysis tools to benefit from multiple and diverse tools [111, 117, 146, 147], and rank the merged alarms using other techniques. We have included them in this category as all those techniques deal with postprocessing of alarms generated by multiple static analysis tools.

Out of the eight primary studies in this sub-category, six studies are identified through the extended literature search. This indicates that the topic of combining results of multiple static analysis tools has recently (in the last three years) started attracting interest from the research community.

Merits and Shortcomings. The approaches in this sub-category allow results of different tools to be validated and in turn rank alarms. However, they require analyzing the same code by multiple tools, and this increases the code analysis time. Therefore, these approaches mostly are not applicable when static analysis tools are part of an IDE: they are used to analyze code as it is being written.

5.2.5 Other Techniques. A few other techniques used to rank alarms include the following:

- Static computation of execution likelihood of the program points at which alarms are reported, also has been used by Boogerd and Moonen [18] for ranking alarms.
- Liang et al. [107] have introduced a novel Expressive Defect Pattern Specification Notation (EDPSN) to define a resource-leak defect pattern more precisely, and have used it to prioritize resource leaks.
- Heo et al. [68] have proposed a technique to rank alarms generated on evolving code. They compute a graph that concisely and precisely captures differences between the derivations of alarms generated before and after the change. Later, they perform Bayesian inference, i.e., statistical inference in which Bayes' theorem is used to update the probability for a hypothesis as more evidence or information becomes available. The Bayesian inference is performed on the graph, which enables to rank alarms by likelihood of relevance to the change.

5.3 Pruning of Alarms

This category of approaches corresponds to classifying alarms into two classes, *actionable* and *non-actionable*. The classification of an alarm is based on the fact that alarms which are not acted upon by the user are seen as false positives and pruned, i.e., not reported to the user. Therefore, these approaches classify alarms depending on the likelihood of the user acting upon the alarms. As the pruned alarms are not guaranteed to be false positives, are approach can result in false negatives. This category is further organized based on the techniques employed to achieve pruning.

5.3.1 Machine Learning-based Pruning. Several studies, such as [2, 61, 65, 183, 185] have employed machine learning to differentiate between actionable and non-actionable alarms. For example, Hanam et al. [61] have achieved a binary classification by finding alarms with similar patterns, where the patterns are identified based on the code surrounding the alarms. Machine learning has been employed to account for semantic and syntactic differences during the identification of patterns. Yüksel and Sözer [185] have evaluated 34 machine learning algorithms in their study using 10 different artifact characteristics. The plentitude of studies that use machine learning to prune alarms [2, 61, 65, 67, 87, 98, 118, 141, 166, 183, 185, 191] suggests popularity of the approach among researchers.

Merits and Shortcomings. Machine learning can help to identify the patterns of code in which alarms are errors (or false positives). Thus, the ML-based alarms postprocessing approaches have been found to be effective. However, training the needed ML models requires traning data having a large number of samples and the appropriate code patterns present. Preparing such training data is a challenge. Moreover, scalability of these ML models to very large code base is still a concern.

5.3.2 Computation of Delta Alarms. The approaches in this sub-category reduce alarms generated during analysis of evolving software by pruning alarms that repeat across versions. Techniques employing this approach apply various analyses to identify (1) alarms that are newly generated as compared to alarms on the previous code version, and (2) repeated alarms which are impacted by the code changes. Alarms reported after applying these techniques are called *delta alarms*.

The techniques proposed to compute delta alarms [30, 93, 109, 158, 168] vary in the methods they use for computation. For example, Spacco et al. [158] have identified newly generated alarms as compared to the previous version, by matching alarms through two approaches: *pairing* and *alarm signatures*. Chimdyalwar and Kumar [30] have proposed an approach to prune repeated alarms generated on evolving software systems. The pruning is achieved by performing an impact analysis—analyzing the impact of changes made between the two successive versions on the alarms—and suppressing alarms that are not impacted by the changes.

Logozzo et al. [109] have introduced a new static analysis technique, called *verification modulo mersions*, to reduce the number of alarms while providing sound semantic guarantees. The proposed technique first extracts semantic environment conditions—sufficient or necessary conditions—from a base program (previous version) and alarms generated on it. Then, it uses the extracted conditions to instrument a new version. Later, it verifies the instrumented code, which reduces alarms generated on the new version.

Merits and Shortcomings. The approaches in this sub-category help by reporting only the alarms that are impacted by code changes between two subsequent versions. Therefore, these approaches are suitable to analyze the code after each code commit. However, these approaches are not applicable when the code to be analyzed is first/initial version or the static analysis tools are part of an IDE.

5.3.3 Other Techniques. Following are a few other techniques proposed to prune alarms.

- Statistical models are used by Ruthruff et al. [152] to identify and prune non-actionable alarms.
- Das et al. [38] have constrained the analysis verifier to report alarms only when no acceptable environment specification (specified through a vocabulary) exists to prove the assertion.
- Chen et al. [26] have pruned alarms corresponding to data-races through thread specialization: distinguishing the threads statically by assigning IDs to threads and fixing their number.

5.4 Automated False Positives Elimination

In this approach, more precise analysis techniques like model checking and symbolic execution are used to identify and eliminate false positives from alarms. An assertion is generated corresponding to each alarm and it is verified using tools based on model checking [29, 37, 143, 167, 184] or symbolic execution [7, 49, 57, 59, 83, 101]. The approaches in this category are more precise as compared to the other approaches, as they *precisely eliminate* false positives from alarms without any user intervention. However, the postprocessing of alarms in this approach generally faces the issues of non-scalability and poor performance due to the state space problem associated with the more precise analysis techniques.

We organize this category into sub-categories based on the techniques used in AFPE: model checking, symbolic execution, and deductive verification. As the approaches in these sub-categories share merits and shortcomings, we discuss them jointly in Section 5.4.4.

5.4.1 Model Checking-based AFPE. In this approach, model checkers such as CBMC [23] are used to eliminate false positives. We partition the approaches in this sub-category into two parts depending on whether they address the *non-scalability* or *poor-efficiency* issues related to the AFPE process.

Note that other combinations of static analysis and model checking have been proposed in the literature [20, 48, 77], where these two techniques iteratively exchange information. We treat this approach differently from false positives elimination, because the aim of this combination is to improve precision of static analysis and improving the analysis precision is out of scope of postprocessing of alarms (Section 3.1.1).

Achieving Scalability. Post et al. [143] have proposed an incremental approach, called *context expansion*, to use a model checker in a more scalable way. In this approach, verification of assertion(s) starts from the function containing the assertions, and then the verification context is gradually incremented to the direct and indirect callers of the function. That is, the verification is started with the minimal context and the context is expanded later on a need basis. This approach also has been observed to be beneficial by other studies [37, 126].

Program slicing [163] also has been another commonly used technique to reduce the state space, and in turn, achieve scalability [29, 37]. On similar lines, a notion of abstract programs has been proposed by Valdiviezo et al. [167] to achieve scalability of model checkers.

Improving Efficiency. The model checking-based AFPE has been found to have poor efficiency due to (1) a large number of alarms that need to be processed, (2) multiple model checking calls for a single assertion due to the context expansion [143], and (3) considerable amount of time (on an average 3 to 5 minutes[5]) that model checking usually takes. To address this issue, i.e., to improve efficiency of AFPE, different techniques have been proposed. For example, Muske et al. [125, 126] have proposed static analysis-based techniques to predict outcome of a given model checking call. The predictions are used to reduce the number of model checking calls and thus, improve AFPE efficiency. Darke et al. [28] partition the generated assertions into disjoint groups based on the data and control flow characteristics, and verify assertions in one group at a time. Wang et al. [173] have used program slicing to improve efficiency of model checking-based AFPE.

5.4.2 Symbolic Execution-based AFPE. In this sub-category, symbolic execution is used to eliminate false positives [83, 101, 186]. In symbolic execution, instead of supplying the concrete inputs to a program (e.g. numbers), symbols representing arbitrary values are supplied, and the values of program variables are represented with symbolic expressions [84]. To address the issue of too many execution paths during symbolic execution, several studies use a variant of symbolic execution, called dynamic symbolic execution (or concolic execution) [7, 49, 57, 59, 139]. This variant involves running a symbolic execution along with a concrete one.

5.4.3 SMT Solvers/Deductive Verification-based AFPE. In this sub-category, SMT solvers [40, 120] or deductive verification (also called *theorem proving*) [50] are used to eliminate false positives [53, 131, 132, 177]. For example, Nguyen et al. [131, 132] use deductive verification to eliminate false positives. In the studies that use SMT solvers for AFPE [53, 177], for each alarm, constraints are generated that represent the conditions under which the alarm is an error. Then, the constraints are checked using a SMT solver to determine their satisfiability. When the constraints are found to be unsatisfiable, the alarm is identified as a false positive and eliminated.

5.4.4 Merits and Shortcomings of AFPE Approaches. The AFPE approaches, discussed in the three sub-categories—model checking-based AFPE, symobolic execution-based AFPE, and SMT Solvers-based AFPE—are automatic and more precise as compared to the other approaches to reduce the number of alarms. However, these approaches usually do not scale to very large systems and have poor efficiency. Other techniques, like program slicing, used for AFPE scalability increase the overall time taken. These approaches are not suitable when the static analysis tools are a part of an IDE, which analyze the code being written/updated on the fly and report alarms instantaneously.

[5] This time taken includes the time taken to generate program slices before the assertion is verified by a model checker.

5.5 Combination of Static and Dynamic Analyses

As a general theme of approaches in this category, static analysis alarms are checked using dynamic analysis if they are true errors and the test cases witnessing failures are reported as error scenarios to the user. This combination requires executing the programs, which is usually absent in static analysis. A few of these studies adopting this combination approach are described below.

Csallner et al. [34] have combined static analysis and concrete test-case generation (Check-n-Crash tool), where a constraint solver is used to derive specific instances of abstract error conditions identified by a static checker (ESC/Java). Later, actual test cases exhibiting error scenarios uncovered by true alarms are presented to the users. As an advancement to this approach, Csallner et al. [35] have used a three step approach, consisting of dynamic inference, static analysis, and dynamic verification (DSD-Crasher tool). The processing in the approach includes (a) inferring likely program invariants using dynamic analysis, (b) using the invariants as assumptions during the static analysis step, and (c) generating test cases that validate true alarms.

Program slicing also has been used for the efficiency of techniques employing this approach: confirming/rejecting more number of alarms in a given time [25]. The efficiency is achieved by reporting more precise error information on simpler programs having shorter program paths and showing values for useful variables only. This reporting reduces the alarms analysis and correction time by the tool users.

Li et al. [103] have proposed a concept of residual investigation—a dynamic analysis serving as the runtime agent of a static analysis—for checking if an alarm is likely to be true. The novelty of the proposed approach lies in predicting errors in executions, which are not actually observed. This predictive nature of their approach is of significant advantage when generation of test cases is hard for very large and complex programs [103].

Merits and Shortcomings. As compared to the other approaches, the approaches in this category present actual error scenarios for true alarms. However, as opposed to static analysis, these approaches require support for executing the programs (e.g., test cases, and run-time environment).

5.6 Simplification of Manual Inspection

The approaches in this category aim to simplify manual inspection of alarms by supporting users during the inspection process. Based on the methods used for the simplification, we organize the approaches into the seven sub-categories described below.

5.6.1 Semi-automatic Alarm Inspection. This sub-category relates to supporting semi-automatic inspection of alarms. For example, to help the user in inspection of alarms by making the inspection more automatic, Rival [148] has enhanced semantic slicing (i.e., computation of precise abstract invariants for a set of erroneous traces) with information about abstract dependences. An abstract dependence is a dependence that can be observed by looking at abstractions of the values of the variables only. In another study, Rival [149] has proposed a framework for semi-automatic inspection of alarms. In this framework, an initial static analysis is refined into an approximation of a subset of traces that actually lead to an error. Later, a combination of forward and backward analyses is used to prove whether this set is empty. If this set is proved to be empty, the alarm is concluded as a false positive.

As another example, Zhu et al. [193] have proposed a novel approach that combines demand-driven analysis and inter-procedural data flow analysis. Using the combined analyses inter-procedural paths are generated to help the user inspect alarms automatically.

Merits and Shortcomings. The approaches in this sub-category provide support for semi-automatic inspection of alarms and simplify the inspection process. However, implementing these approaches requires implementing data flow analyses, and the degree of simplification achieved depends on the precision and scalability of those analyses.

5.6.2 Feedback-based Manual Inspection. A few studies have been found to capture user-feedback to simplify manual inspection of alarms. For example, Mangal et al. [114] have formulated user-guided program analysis to shift decisions about the kind and degree of approximations to apply in an analysis from the analysis writer to the analysis user. In the proposed analysis approach, user feedback about which analysis results are liked or disliked is captured and the analysis is re-run [114]. This approach uses soft rules to capture the user preferences and allows users to control both the precision and scalability of the analysis.

Sadowski et al. [153] have proposed a program analysis platform to build a data-driven ecosystem around static analysis. The platform is based on a feedback loop between the users of static analysis tool(s) and writers of those tool(s). The feedback loop is towards simplifying inspection of alarms reported by the tools.

Merits and Shortcomings. These approaches allow users to control the reporting of alarms based on the feedback and report only the alarms that are identified as important by the user. However, user involvement is a must during postprocessing of alarms using these approaches.

5.6.3 Checklists-based Manual Inspection. In these approaches, checklists are used to systematically guide users during manual inspection of alarms. Ayewah et al. [8] have proposed use of checklists to enable more detailed review of static analysis alarms. On similar lines, Phang et al. [140] have used triaging checklists to provide systematic guidance to users during manual inspection of alarms. The users follow the instructions on the checklist to answer each question and determine conclusions about the alarms. They also propose that the checklists are designed by tool developers so that, (a) known sources of imprecision in their tools are pointed out, and (b) users are instructed on how to look for those sources of imprecision. Additionally, the checklists are customized to individual alarms so that a minimum number of questions are answered during inspection of an alarm.

Merits and Shortcomings. These approaches provide systematic guidance to users during manual inspection of alarms, and help locate the causes for generation of the alarms and the remediation required. Applying these approaches to all types of alarms, including the ones that are obvious to inspect, may slow down the inspection process.

5.6.4 Alarms-relevant Queries. In the approaches in this sub-category, alarm-specific queries are presented to the user for achieving effective and efficient manual inspection of alarms. For example, Dillig et al. [42] have proposed an approach to semi-automatically classify alarms into errors and false positives by presenting alarm-specific queries to the users. Abductive inference is used to compute small and relevant queries that capture exactly the information needed from user to discharge or validate an alarm. Two types of queries, proof obligation and failure witness queries, are framed, and they are ranked using a cost function so that easy-to-answer queries are presented first to the users.

Zhang et al. [190] have combined a sound but imprecise analysis with precise but unsound heuristics, through user interaction. This combined approach poses questions to the user about the root causes that are targeted by the heuristic. If the user confirms them, only then is the heuristic applied to eliminate the false alarms. Muske and Khedker [127] have proposed cause points analysis. In this analysis, root causes of alarms are identified, and queries generated specific to the causes are presented to the user for reducing the manual inspection effort.

Merits and Shortcomings. The approaches in sub-category help significantly reduce the inspection effort, because the alarm-specific queries avoid user's redundant effort in locating the cause points and constraints to be checked on them. In these approaches, user is mostly required to answer the queries as yes or no. The techniques used to automatically generate the queries may not scale to very large systems.

5.6.5 Automated Repair. Recent studies have also been aiming at automated repair of alarms. For example, Bavishi et al. [15] have proposed a technique for automatically generating patches for static analysis violations by learning from examples. Aiming at improving usability of static analysis tools, Marcilio et al. [115] automatically provide fix suggestions, that is modifications to the source code that make it compliant with the rules checked by the tools. As another example of this approach, Xue et al. [179] propose a history-driven approach to automatically fix code quality issues detected by static analysis tools, by utilizing the fixing knowledge mined from the change history in the code repositories.

Merits and Shortcomings. These approaches reduce user's effot required to fix alarms when they are identified as errors. The effort reduction is because, instead of identifying the fixes for alarms, the user is required only to accept or reject the automatically generated fixes. However, these approaches do not help in identifying alarms that are false positives or can be safely eliminated.

5.6.6 Usage of Novel User-interfaces/Visualization Tools. This sub-category deals with usage of code navigation/visualization tools that simplify the code traversals performed by the user while inspecting alarms manually [5, 33, 70]. For example, Phang et al. [79] have presented a novel user interface toolkit, called path projection, to help users to visualize, navigate, and understand program paths during the inspection. Parnin et al. [138] have used a catalogue of lightweight visualizations to help users during inspection of alarms. In their study, a simple light-weight visualization is designed for each alarm. Buckers et al. [21] have proposed UAV (Unified ASAT Visualizer) that provides an intuitive visualization, enabling developers, researchers, and tool creators to compare the complementary strengths and overlaps of different static analysis tools applicable for Java programs. The UAV's enriched treemap and source code views provide its users with a seamless exploration of the alarm distribution from a high-level overview down to the source code.

Merits and Shortcomings. The approaches in this sub-category help user when the code that the user should inspect manually, is large and complex. However, user involvment is a must.

5.6.7 Other Techniques. Following are a few other techniques proposed to simplify inspection of alarms.

- Arai et al. [6] have explored a gamification approach and proposed a novel gamified tool for motivating developers to inspect alarms. The tool proposed calculates scores based on the alarms inspected by each developer or team. On similar lines, Nguyen Quang Do et al. [133] have proposed to leverage the knowledge of game designers, and to integrate gaming elements into analysis tools to improve their user experience.
- The overflow of information and decisions to be made during manual inspection of alarms can be tiring and cause stress symptoms to the users. To fight the stress, Ostberg and Wagner [136] have proposed to use *salutogenesis model* that has been used in health care.
- Aiming to evaluate and improve quality of reports of static analysis tools, Menshchikov and Lepikhin [119] generalize the tool output messages and explore ways to improve reliability of comparison results. To this end, they introduce informational value as a measure of report quality with respect to 5Ws (What, When, Where, Who, Why) and 1H (How To Fix) questions.

6 DISCUSSION

In this section, we discuss merits and shortcomings of the approaches identified, present guidelines for selection of the approaches, and provide directions for future work. We then discuss the threats to validity.

6.1 Summary of Merits and Shortcomings of the Approaches

In Table 3 we summarize merits and shortcomings of the identified approaches for postprocessing of alarms. Clustering of alarms (Section 5.1) helps to reduce effort required to manually inspect alarms. Unlike sound clustering, unsound clustering may result in false negatives. When alarms are ranked (Section 5.2), the alarms that are more likely to be errors than other alarms get inspected early during manual inspection. However, all the ranked alarms need to be inspected, otherwise the inspection can result in false negatives. Pruning of alarms, i.e., classification of alarms into actionable and non-actionable alarms and suppressing the non-actionable alarms (Section 5.3), reduces the number of alarms shown to the user, however the suppression of alarms can result in false negatives. Among the approaches that prune alarms based on relative correctness (Section 5.3.2), computation of delta alarms based on impact analysis [30] and verification modulo versions [109] can be used to prune alarms without resulting in false negatives.

Approaches that eliminate false positives automatically (Section 5.4) are more precise as compared to the other approaches aiming at reducing the number of alarms. However non-scalability and poor efficiency are two major concerns associated with those approaches. Approaches that combine static analysis with dynamic analysis (Section 5.5) help to present error scenarios for true alarms. However, these approaches require the code to be executable and ensuring that test cases will execute the statements associated with alarms. Moreover, it is very hard to identify an alarm as a false positive using these approaches. The approaches that simplify manual inspection of alarms (Section 5.6) can provide significant aid to users during manual inspection, however user involvement is required during postprocessing of alarms.

Table 3 shows that the categories of approaches are complementary: shortcomings of one approach are merits of some other approaches and they can complement each other. Thus, their combinations are possible to obtain better postprocessing results (Section 6.3).

6.2 Guidelines for Selection of Approaches

Postprocessing of alarms differs greatly depending on whether the purpose of the analysis is to find bugs (*bug finding*) or to prove absence of bugs of certain types (*code proving*). For example, approaches to postprocess alarms generated by code proving tools on safety-critical applications would be different from approaches applicable for postprocessing alarms generated by bug finding tools on applications that are not safety-critical. Approaches applicable for postprocessing of alarms generated by code proving tools can be applied to alarms generated by *bug finding tools*, but the converse is not true. If the aim of the analysis is code proving, alarms can be postprocessed using sound clustering (Section 5.1.1), ranking (Section 5.2), AFPE (Section 5.4), and simplification of manual inspection (Section 5.6), because postprocessing alarms using approaches in these categories does not result in false negatives[6]. Moreover, if an application being analyzed is evolving and code proving is the analysis purpose, approaches that prune alarms based on impact analysis [30] and verification modulo versions [109] also can be used. If the analysis purpose is bug finding, alarms can be postprocessed using approaches from all categories and sub-categories.

We observe that, for effective postprocessing of alarms, the postprocessing needs to take into account the type of applications being analyzed. For example, techniques which are suitable to process

[6] No alarm is removed/pruned unless it is guaranteed be false either automatically or by the tool-user.

No.	Category	Merits	Shortcomings
A.	Clustering	In sound clustering, mostly dominant alarms need to be inspected while skipping inspection of the other alarms, whereas in unsound clustering, the group-wise inspection of alarms reduces inspection effort.	Unsound clustering may result in false negatives. Reduction in the number of alarms (by sound clustering) depends on the percentage of alarms identified as dominant.
B.	Ranking	During manual inspection of alarms, the alarms that are more likely to be errors get inspected early, and this approach does not result in false negatives.	It requires inspecting all the reported alarms.
C.	Pruning	Only the actionable alarms are to be inspected, while the other alarms being identified as non-actionable are not reported to the user.	This pruning approach may result in false negatives as a pruned alarm cannot be guaranteed to be a false positive (except the reliable techniques that compute delta alarms [30, 109]).
D.	AFPE	It is automatic and more precise as compared to the other approaches to reduce the number of alarms.	Techniques employing this approach usually face issues related to non-scalability and poor efficiency.
E.	Combination of static and dynamic analyses	It presents error scenarios for true alarms.	It requires support for executing the programs (e.g., test cases, and run-time environment) which is usually absent during static analysis process.
F.	Simplification of manual inspection	It provides significant aid to the users during manual inspection.	User involvement is a must.

Table 3. Merits and shortcomings of the categories of approaches for postprocessing of alarms.

alarms generated on evolving code are not suitable to process alarms generated on partitioned-code or family of products, and vice versa. When an application to be analyzed belongs to multiple types, it would be more beneficial if more than one approach is applied, because combination of different approaches will help to reduce more alarms or simplify inspection of alarms further. For example, analyzing partitioned-code of an evolving application would require combining techniques that take into account the nature of multiple versions (pruning based on relative correctness [30, 109]) and multiple partitions (clustering based on common points of interest [121]).

The diversity of applications in terms of programming languages and coding practices induce different requirements when it comes to postprocessing of alarms and make the postprocessing much more challenging. For example, precise analysis of C programs suffers from imprecision due to pointers and programmatic means of accessing hardware (memory) registers. Precise analysis of C++/Java programs requires considering such features as aliases, virtual functions, (multiple) inheritance, reflection and templates. Postprocessing of alarms in general requires re-analysis of the program, and hence, all those characteristics of the language need to be considered in the selection of postprocessing approaches. These characteristics can affect the postprocessing results.

From the evaluations of the postprocessing approaches, e.g., clustering approaches [99, 100, 123, 129, 130, 187], we find that the postprocessing results on applications vary considerably even if the applications are written in the same programming language and belong to the same domain. For example, the number of alarms that are similar to and get identified by sound clustering techniques as followers of other alarms is found to vary considerably among the C applications. This indicates that some applications have more follower alarms, so that the clustering approaches achieve higher reduction as compared to the other applications. Thus, instead of postprocessing alarms using a single chosen approach for all applications, more appropriate approach(es) should be selected based on occurrences of patterns of alarms present in the application being analyzed.

6.3 Directions of Future Work

As discussed in Section 6.1, the approaches identified are orthogonal, and hence they can be combined to obtain better postprocessing results. We observe that possible combinations of the approaches are not widely studied or evaluated. A few studies [90, 125] consider the combinations of the approaches and find the combinations to be promising. Thus, exploring the possible combinations of these various approaches can be a future research direction to postprocess alarms more effectively. For example, only the dominant alarms identified through sound clustering (Section 5.1.1) can be ranked (Section 5.2) or pruned (Section 5.3). In another example, pruning of alarms followed by AFPE (Section 5.4) can help each other: AFPE eliminates false positives from the actionable alarms resulting after pruning, and processing only the actionable alarms (a subset of alarms generated) reduces the number of alarms to be processed by the AFPE techniques.

Moreover, combinations of the approaches can be implemented with two strategies: sequencing the approaches one after the other (pipelining), and running them in parallel. The positioning of approaches in the combinations with pipelining can vary depending on the requirements in practice. For example, choice for the first approach to be implemented, between pruning and AFPE when they are to be combined, can be made based on total time available for processing the alarms. The other strategy, parallelization of the approaches can help in enhancing confidence about false positives and errors. For example, the results obtained in isolation from approaches like ranking, pruning, and combination of static and dynamic analyses, can be merged together to increase confidence about alarms that are more likely to be false positives or errors. More research needs to be conducted to identify effective sequential and parallel combinations of approaches.

Since an approach to postprocess alarms can be implemented using several orthogonal techniques, e.g., the techniques to implement pruning of alarms are orthogonal, multiple techniques can be implemented and run in parallel to process alarms as per the chosen approach. We believe implementing such combinations of approaches and techniques in practice however will increase the analysis time multi-fold. Hence, reducing the analysis time is equally important, and performance of such combinations also needs to be studied while studying advantages and disadvantages of the combinations. One approach to address this problem of the increased analysis time can be to *quickly predict* the improvement that a time-consuming technique can provide before the technique is applied. To this end, further research needs to be conducted to identify criteria for those techniques, that will help to predict postprocessing results on a given application. Based on time availability and prediction results of the applicable techniques, a subset of techniques can be selected from a set of applicable techniques.

Considering the advancement in machine-learning based techniques and their application to process alarms (Section 5.3.1), we believe that machine learning can be used to select a subset of techniques from a set of applicable techniques, which are more suitable on a given application compared to the other techniques. To this end, applicability of machine learning needs to be studied to learn the types (structures) of applications on which each of the techniques provides better results, and then to select the most suitable technique(s) for a given new application.

We observe that complexity of the problem of alarms' postprocessing varies as per factors like diversity of goals static analysis can be applied for, applications being analyzed, and programming languages. For example, suppression of an alarm that is an error, is not allowed when the goal of static analysis is code proving whereas this restriction generally is not applicable when the purpose is bug finding. Therefore, these goals impact how the alarms are to be processed. Moreover, a programming language also impacts complexity of postprocessing of alarms, e.g., as JavaScript is an untyped language (unlike C, C++, or Java), postprocessing of alarms generated on JavaScript applications will be more complex as compared to postprocessing of alarms on C applications. Therefore, to improve postprocessing of alarms, research needs to be continued along multiple lines to support different factors like the analysis purposes, different programming paradigms, and types of applications.

6.4 Threats to Validity

6.4.1 Construct Validity. Our literature study has focused on understanding what techniques have been proposed in scientific literature for postprocessing of alarms. Threats to construct validity concern how accurately we operationalize the notion of *scientific papers discussing postprocessing of alarms*, i.e. how we collect research papers that propose approaches for postprocessing of alarms.

In a keywords-based literature search, the selection of keywords (i.e., search strings used) affects the papers identified through the search [66, 86]. The keywords selected in our study are based on the author's experience in reading and writing research papers on the topic of postprocessing of alarms [123–127]. This list of search strings used may not be complete, and hence a relevant paper might be missed. Moreover, since reviewing all the results for each search string may not be practically possible, for each search string we reviewed only the first 150 search results. As a result, we might have missed relevant papers which appeared later in the results list. To mitigate these two issues, we performed snowballing by creating the start set from the relevant papers collected through the keywords-based search. The snowballing helped to identify relevant papers which were missed by the search.

Ideally, a literature search is to be performed by multiple researchers who have expertise in the topic under study: postprocessing of alarms. Getting such multiple experts is difficult because such experts are found to be rare, and the experts require to spend considerable amount of time in the search and reviewing the relevant papers. Therefore, the complete search, extraction of data, and categorization of approaches is performed solely by the author without involving additional experts. This might lead to subjective bias in the identification of relevant papers and data extraction. Moreover, the selected inclusion and exclusion criteria may introduce attrition bias [66]. To reduce subjectivity threats inherent in application of inclusion and exclusion criteria in literature studies we have performed an additional quality assurance step. As shown in 2, 46/12600 of the papers found by the keywords-based search have been deemed relevant, i.e., less than 1%. Hence, the second author has manually labeled (relevant/not relevant) 16 papers randomly selected out of 12600 corresponding to the commonly used confidence level (95%) and confidence interval (5%). In all cases the results of the labeling of the second author agreed with the labeling of the first author.

6.4.2 Internal Validity. Threats to internal validity concern the extent to which the observations are grounded in the data collected from the papers identified. Since the data extraction and categorization of the approaches are performed by the author without involving additional experts, the categorization of the approaches (Figure 1) and the observations made from them (Table 3 and Section 6) might have been affected by the subjectivity bias. Comparing findings from our study with the similar studies is one way to validate the findings. However, no such studies exist. The existing studies about static analysis tools and alarms [39, 45, 66] have different goals and are not immediately comparable to our study.

6.4.3 External Validity. Threats to external validity concern the extent to which results of the study generalize beyond the sample studied to the entire population. In our study, since the sample studied (i.e., 130 papers we have studied) is the same as the population (i.e., all papers ever published about postprocessing of alarms), threats to external validity are not applicable.

7 CONCLUSION AND FUTURE WORK

In this article, we have studied approaches proposed in the literature to postprocess alarms generated by static analysis tools. We conducted systematic literature search, by combining keywords-based database search and snowballing, to collect research papers that propose techniques for postprocessing of alarms. Through the literature search, we identified 130 papers relevant to the topic. We reviewed the papers, and extracted and categorized the approaches proposed by them. Our categorization shows that, six main categories of the approaches, namely clustering, ranking, pruning, AFPE, combination of static and dynamic analyses, and simplification of manual inspection, have been proposed for postprocessing of alarms.

During the categorization, wherever appropriate, we have further categorized the main categories into multiple sub-categories depending on the techniques used to implement those approaches. We have summarized the merits and shortcomings of these categories (Section 6) to assist users and designers/developers of static analysis tools to make informed choices. Several of the identified categories—sound clustering, ranking, AFPE, simplification of manual inspection—can help static analysis tools that are used for code proving.

The categorized postprocessing approaches, being complementary, provide an opportunity to combine them. We observe that, the identified (sub)-categories of approaches can be combined together in several ways, and feasibility of the different combinations, their advantages and disadvantages, however, need to be studied. Moreover, more research needs to be conducted to quickly predict results of the applicable postprocessing approaches on a given application, so that alarms on that application can be processed by the most suitable approaches. On similar lines, research needs to be conducted for selecting technique(s) that are more suitable to implement an approach chosen for a given application.

REFERENCES

[1] Ashish Aggarwal and Pankaj Jalote. Integrating static and dynamic analysis for detecting vulnerabilities. In *International Computer Software and Applications Conference*, pages 343–350. IEEE, 2006.

[2] Enas A. Alikhashashneh, Rajeev R. Raje, and James H. Hill. Using machine learning techniques to classify and predict static code analysis tool warnings. In *International Conference on Computer Systems and Applications*, pages 1–8. IEEE, 2018.

[3] Simon Allier, Nicolas Anquetil, Andre Hora, and Stephane Ducasse. A framework to compare alert ranking algorithms. In *Working Conference on Reverse Engineering*, pages 277–285. ACM, 2012.

[4] Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. A survival analysis-based prioritization of code checker warning: A case study using PMD. In *International Conference on Big Data, Cloud Computing, and Data Science Engineering*, pages 69–83. Springer, 2019.

[5] Paul Anderson, Thomas Reps, Tim Teitelbaum, and Mark Zarins. Tool support for fine-grained software inspection. *IEEE software*, 20(4):42–50, 2003.

[6] Satoshi Arai, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. A gamified tool for motivating developers to remove warnings of bug pattern tools. In *International Workshop on Empirical Software Engineering in Practice*, pages 37–42. IEEE, 2014.

[7] Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. Using targeted symbolic execution for reducing false-positives in dataflow analysis. In *International Workshop on State Of the Art in Program Analysis*, pages 1–6. ACM, 2015.

[8] Nathaniel Ayewah and William Pugh. Using checklists to review static analysis warnings. In *International Workshop on Defects in Large Software Systems*, pages 11–15. ACM, 2009.

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

[10] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 1–8. ACM, 2007.

[11] Deepika Badampudi, Claes Wohlin, and Kai Petersen. Experiences from using snowballing and database searches in systematic literature studies. In *International Conference on Evaluation and Assessment in Software Engineering*, pages 17:1–17:10. ACM, 2015.

[12] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[13] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. In *International Static Analysis Symposium*, pages 337–354. Springer, 2003.

[14] Titus Barik, Yoonki Song, Brittany Johnson, and Emerson Murphy-Hill. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration. In *International Conference on Software Maintenance and Evolution*, pages 211–221. IEEE, 2016.

[15] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 613–624. ACM, 2019.

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

[17] Sam Blackshear and Shuvendu K. Lahiri. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *Conference on Programming Language Design and Implementation*, pages 209–218. ACM, 2013.

[18] Cathal Boogerd and Leon Moonen. Prioritizing software inspection results using static profiling. In *International Workshop on Source Code Analysis and Manipulation*, pages 149–160. IEEE, 2006.

[19] Guillaume Brat and Arnaud Venet. Precise and scalable static program analysis of NASA flight software. In *Aerospace Conference*, pages 1–10. IEEE, 2005.

[20] Guillaume Brat and Willem Visser. Combining static analysis and model checking for software analysis. In *International Conference on Automated Software Engineering*, pages 262–269. IEEE, 2001.

[21] Tim Buckers, Clinton Cao, Michiel Doesburg, Boning Gong, Sunwei Wang, Moritz Beller, and Andy Zaidman. UAV: Warnings from multiple automated static analysis tools at a glance. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 472–476. IEEE, 2017.

[22] Aji Ery Burhandenny, Hirohisa Aman, and Minoru Kawahara. Investigation of coding violations focusing on authorships of source files. In *International Conference on Applied Computing and Information Technology/International Conference on Computational Science Intelligence and Applied Informatics/International Conference on Big Data, Cloud Computing, Data Science*, pages 248–253. IEEE, 2017.

[23] CBMC. http://www.cprover.org/cbmc/.

[24] George Chatzieleftheriou and Panagiotis Katsaros. Test-driving static analysis tools in search of C code vulnerabilities. In *Computer Software and Applications Conference Workshops*, pages 96–103. IEEE, 2011.

[25] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *Symposium on Applied Computing*, pages 1284–1291. ACM, 2012.

[26] Chen Chen, Kai Lu, Xiaoping Wang, Xu Zhou, and Li Fang. Pruning false positives of static data-race detection via thread specialization. In *International Workshop on Advanced Parallel Processing Technologies*, pages 77–90. Springer, 2013.

[27] Ping Chen, Hao Han, Yi Wang, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. IntFinder: Automatically detecting integer bugs in x86 binary program. In *International Conference on Information and Communications Security*, pages 336–345. Springer, 2009.

[28] Bharti Chimdyalwar and Priyanka Darke. Statically relating program properties for efficient verification. In *International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 99–103. ACM, 2018.

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

[30] Bharti Chimdyalwar and Shrawan Kumar. Effective false positive filtering for evolving software. In *India Software Engineering Conference*, pages 103–106. ACM, 2011.

[31] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *International Conference on Automated Software Engineering*, pages 332–343. ACM, 2016.

[32] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[33] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *European Symposium on Programming*, pages 21–30. Springer, 2005.

[34] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: Combining static checking and testing. In *International Conference on Software Engineering*, pages 422–431. ACM, 2005.

[35] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology*, 17(2):1–37, 2008.

[36] Priyanka Darke, Bharti Chimdyalwar, Avriti Chauhan, and R. Venkatesh. Efficient safety proofs for industry-scale code using abstractions and bounded model checking. In *International Conference on Software Testing, Verification and Validation*, pages 468–475. IEEE, 2017.

[37] Priyanka Darke, Mayur Khanzode, Arun Nair, Ulka Shrotri, and R. Venkatesh. Precise analysis of large industry code. In *Asia-Pacific Software Engineering Conference*, pages 306–309. IEEE, 2012.

[38] Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In *International Conference on Computer Aided Verification*, pages 324–342. Springer, 2015.

[39] Vinicius Rafael Lobo de Mendonca, Cassio Leonardo Rodrigues, Fabrízzio Alphonsus Alves de Melo Nunes Soares, and Auri Marcelo Rizzo Vincenzi. Static analysis techniques and tools: A systematic mapping study. In *International Conference on Software Engineering Advances*, pages 72–78. IARIA XPS Press, 2013.

[40] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In *International Conference on Computer Aided Verification*, pages 20–36. Springer, 2007.

[41] Ewen Denney and Steven Trac. A software safety certification tool for automatically generated guidance, navigation and control code. In *Aerospace Conference*, pages 1–11. IEEE, 2008.

[42] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In *Conference on Programming Language Design and Implementation*, pages 181–192. ACM, 2012.

[43] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time static analysis. In *International Symposium on Software Testing and Analysis*, pages 307–317. ACM, 2017.

[44] Lisa Nguyen Quang Do, James Wright, and Karim Ali. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering*, 2020.

[45] Frank Elberzhager, Jürgen Münch, and Vi Tran Ngoc Nha. A systematic mapping study on the combination of static and dynamic quality assurance techniques. *Information and Software Technology*, 54(1):1–15, 2012.

[46] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. Technical report, STANFORD UNIV CA COMPUTER SYSTEMS LAB, 2000.

[47] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *International conference on formal verification of object-oriented software*, pages 10–30. Springer, 2010.

[48] Ansgar Fehnker and Ralf Huuck. Model checking driven static analysis for the real world: designing and tuning large scale bug detection. *Innovations in Systems and Software Engineering*, 9(1):45–56, 2013.

[49] Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and Marie-Laure Potet. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In *Workshop on Software Security, Protection, and Reverse Engineering*, page 2. ACM, 2016.

[50] Jean-Christophe Filliâtre. Deductive software verification. Springer, 2011.

[51] Lori Flynn, William Snavely, David Svoboda, Nathan VanHoudnos, Richard Qin, Jennifer Burns, David Zubrow, Robert Stoddard, and Guillermo Marce-Santurio. Prioritizing alerts from multiple static analysis tools, using classification models. In *International Workshop on Software Qualities and Their Dependencies*, pages 13–20. ACM, 2018.

[52] Zachary P. Fry and Westley Weimer. Clustering static analysis defect reports to reduce maintenance costs. In *Working Conference on Reverse Engineering*, pages 282–291. IEEE, 2013.

[53] Mikhail R. Gadelha, Enrico Steffinlongo, Lucas C. Cordeiro, Bernd Fischer, and Denis Nicole. SMT-based refutation of spurious bug reports in the Clang Static Analyzer. In *International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 11–14. IEEE, 2019.

[54] Fengjuan Gao, Linzhang Wang, and Xuandong Li. BovInspector: automatic inspection and repair of buffer overflow vulnerabilities. In *International Conference on Automated Software Engineering*, pages 786–791. IEEE, 2016.

[55] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. DyTa: dynamic symbolic execution guided with static verification results. In *International Conference on Software Engineering*, pages 992–994. ACM, 2011.

[56] Alexander Gerasimov, Sergey Vartanov, Mikhail Ermakov, Leonid Kruglov, Daniil Kutz, Alexander Novikov, and Seryozha Asryan. Anxiety: a dynamic symbolic execution framework. In *2017 Ivannikov ISPRAS Open Conference (ISPRAS)*, pages 16–21. IEEE, 2017.

[57] Alexander Yu. Gerasimov. Directed dynamic symbolic execution for static analysis warnings confirmation. *Programming and Computer Software*, 44(5):316–323, Sep 2018. Translated by O. Pismenov.

[58] Alexander Yu. Gerasimov and Leonid V. Kruglov. Reachability confirmation of statically detected defects using dynamic analysis. In *Ivannikov Memorial Workshop*, pages 24–31. IEEE, 2018.

[59] Alexander Yu. Gerasimov, Leonid V. Kruglov, Mikhail K. Ermakov, and Sergey P. Vartanov. An approach to reachability determination for static analysis defects with the help of dynamic symbolic execution. In *Programming and Computer Software*, volume 44, pages 467–475. Springer, 2018.

[60] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Automatically refining abstract interpretations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 443–458. Springer, 2008.

[61] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. Finding patterns in static analysis alerts: Improving actionable alert ranking. In *Working Conference on Mining Software Repositories*, pages 152–161. ACM, 2014.

[62] Eric Haugh and Matt Bishop. Testing c programs for buffer overflow vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, page 8. Citeseer, 2003.

[63] Sarah Heckman. Adaptively ranking alerts generated from automated static analysis. *XRDS: Crossroads, The ACM Magazine for Students*, 14(1):1–11, 2007.

[64] Sarah Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *International Symposium on Empirical Software Engineering and Measurement*, pages 41–50. ACM, 2008.

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

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

[67] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-learning-guided selectively unsound static analysis. In *International Conference on Software Engineering*, pages 519–529. IEEE, 2017.

[68] Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. Continuously reasoning about programs using differential Bayesian inference. In *Conference on Programming Language Design and Implementation*, pages 561–575. ACM, 2019.

[69] IK Isaev and DV Sidorov. The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs. *Programming and Computer Software*, 36(4):225–236, 2010.

[70] Raoul Praful Jetley, Paul L. Jones, and Paul Anderson. Static analysis of medical device software using CodeSonar. In *Workshop on Static Analysis*, pages 22–29. ACM, 2008.

[71] Jun-li Jiao, Da-hai Jin, and Ming-nan Zhou. Dominant alarm research and implementation based on static defect detection. *DEStech Transactions on Computer Science and Engineering*, (cimns), 2017.

[72] Jlint. http://artho.com/jlint/.

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

[74] S. C. Johnson. Lint, a C program checker. *Computer Science Technical Report*, 1978.

[75] Saurabh Joshi, Shuvendu K. Lahiri, and Akash Lal. Underspecified harnesses and interleaved bugs. *ACM SIGPLAN Notices*, 47(1):19–30, 2012.

[76] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis. In *International Static Analysis Symposium*, pages 203–217. Springer, 2005.

[77] Maximilian Junker, Ralf Huuck, Ansgar Fehnker, and Alexander Knapp. SMT-based false positive elimination in static program analysis. In *International Conference on Formal Engineering Methods*, pages 316–331. Springer, 2012.

[78] Uday P. Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data Flow Analysis: Theory and Practice*. CRC Press, 2017.

[79] Yit Phang Khoo, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. Path projection for user-centered static analysis tools. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 57–63. ACM, 2008.

[80] Joon-Ho Kim, Myung-Chul Ma, and Jae-Pyo Park. An analysis on secure coding using symbolic execution engine. *Journal of Computer Virology and Hacking Techniques*, 12(3):177–184, 2016.

[81] Sunghun Kim and Michael D. Ernst. Prioritizing warning categories by analyzing software history. In *International Workshop on Mining Software Repositories*, pages 27–27. IEEE, 2007.

[82] Sunghun Kim and Michael D. Ernst. Which warnings should I fix first? In *Joint meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 45–54. ACM, 2007.

[83] Youil Kim, Jooyong Lee, Hwansoo Han, and Kwang-Moo Choe. Filtering false alarms of buffer overflow analysis using SMT solvers. *Information and Software Technology*, 52(2):210–219, 2010.

[84] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[85] Balázs Kiss, Nikolai Kosmatov, Dillon Pariente, and Armand Puccetti. Combining static and dynamic analyses for vulnerability detection: illustration on heartbleed. In *Haifa Verification Conference*, pages 39–50. Springer, 2015.

[86] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

[87] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. In *International Workshop on Machine Learning and Programming Languages*, pages 35–42. ACM, 2017.

[88] Deguang Kong, Quan Zheng, Chao Chen, Jianmei Shuai, and Ming Zhu. ISA: A source code static vulnerability detection system based on data fusion. In *International Conference on Scalable Information Systems*, pages 55:1–55:7. ICST, 2007.

[89] Andrew Kornecki and Janusz Zalewski. Certification of software for real-time safety-critical systems: state of the art. *Innovations in Systems and Software Engineering*, 5(2):149–161, 2009.

[90] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *International Symposium on Foundations of Software Engineering*, pages 83–93. ACM, 2004.

[91] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *International Static Analysis Symposium*, pages 295–315. Springer, 2003.

[92] Rahul Kumar and Aditya V. Nori. The economics of static analysis tools. In *Joint Meeting on Foundations of Software Engineering*, pages 707–710. ACM, 2013.

[93] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *Joint Meeting on Foundations of Software Engineering*, pages 345–355. ACM, 2013.

[94] Davy Landman, Alexander Serebrenik, Eric Bouwers, and Jurgen J. Vinju. Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions. *Journal of Software: Evolution and Process*, 28(7):589–618, 2016.

[95] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. Challenges for static analysis of Java reflection-literature review and empirical study. In *International Conference on Software Engineering*, pages 507–518. IEEE, 2017.

[96] Lucas Layman, Laurie Williams, and Robert St. Amant. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *International Symposium on Empirical Software Engineering and Measurement*, pages 176–185. IEEE, 2007.

[97] Wei Le and Mary Lou Soffa. Path-based fault correlations. In *International Symposium on Foundations of Software Engineering*, pages 307–316. ACM, 2010.

[98] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim, and Shin Yoo. Classifying false positive static checker alarms in continuous integration using convolutional neural networks. In *Conference on Software Testing, Validation and Verification*, pages 391–401. IEEE, 2019.

[99] Woosuk Lee, Wonchan Lee, Dongok Kang, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Sound non-statistical clustering of static analysis alarms. *ACM Transactions on Programming Languages and Systems*, 39(4):1–35, 2017.

[100] Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. Sound non-statistical clustering of static analysis alarms. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 299–314. Springer, 2012.

[101] Hongzhe Li, Taebeom Kim, Munkhbayar Bat-Erdene, and Heejo Lee. Software vulnerability detection using backward trace analysis and symbolic execution. In *International Conference on Availability, Reliability and Security*, pages 446–454. IEEE, 2013.

[102] J. Jenny Li, John Palframan, and Jim Landwehr. SoftWare IMmunization (SWIM) - a combination of static analysis and automatic testing. In *Annual Computer Software and Applications Conference*, pages 656–661. IEEE, 2011.

[103] Kaituo Li, Christoph Reichenbach, Christoph Csallner, and Yannis Smaragdakis. Residual investigation: Predictive and precise bug detection. In *International Symposium on Software Testing and Analysis*, pages 298–308. ACM, 2012.

[104] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of Android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.

[105] Mengchen Li, Yuanjun Chen, Linzhang Wang, and Guoqing Xu. Dynamically validating static memory leak warnings. In *International Symposium on Software Testing and Analysis*, pages 112–122. ACM, 2013.

[106] Guangtai Liang, Ling Wu, Qian Wu, Qianxiang Wang, Tao Xie, and Hong Mei. Automatic construction of an effective training set for prioritizing static analysis warnings. In *International Conference on Automated Software Engineering*, pages 93–102. ACM, 2010.

[107] Guangtai Liang, Qian Wu, Qianxiang Wang, and Hong Mei. An effective defect detection and warning prioritization approach for resource leaks. In *Annual Computer Software and Applications Conference*, pages 119–128. IEEE, 2012.

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

[109] Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. Verification modulo versions: Towards usable verification. In *Conference on Programming Language Design and Implementation*, pages 294–304. ACM, 2014.

[110] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *International Symposium on Code Generation and Optimization*, pages 136–146. IEEE, 2009.

[111] Bailin Lu, Wei Dong, Liangze Yin, and Li Zhang. Evaluating and integrating diverse bug finders for effective program analysis. In *International Conference on Software Analysis, Testing, and Evolution*, pages 51–67. Springer, 2018.

[112] Xutong Ma, Jiwei Yan, Jun Yan, and Jian Zhang. Reorganizing and optimizing post-inspection on suspicious bug reports in path-sensitive analysis. In *International Conference on Software Quality, Reliability and Security*, pages 260–271. IEEE, 2019.

[113] Stephen MacDonell, Martin Shepperd, Barbara Kitchenham, and Emilia Mendes. How reliable are systematic reviews in empirical software engineering? *IEEE Transactions on Software Engineering*, 36(5):676–687, 2010.

[114] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. A user-guided approach to program analysis. In *Joint Meeting on Foundations of Software Engineering*, pages 462–473. ACM, 2015.

[115] Diego Marcilio, Carlo A. Furia, Rodrigo Bonifácio, and Gustavo Pinto. Automatically generating fix suggestions in response to static code analysis warnings. In *International Working Conference on Source Code Analysis and Manipulation*, pages 34–44. IEEE, 2019.

[116] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *International Conference on World Wide Web*, pages 63–74. ACM, 2014.

[117] Na Meng, Qianxiang Wang, Qian Wu, and Hong Mei. An approach to merge results of multiple static analysis tools (short paper). In *International conference on quality software*, pages 169–174. IEEE, 2008.

[118] Qingkun Meng, Chao Feng, Bin Zhang, and Chaojing Tang. Assisting in auditing of buffer overflow vulnerabilities via machine learning. *Mathematical Problems in Engineering*, 2017, 2017.

[119] Maxim Menshchikov and Timur Lepikhin. 5w+ 1h static analysis report quality measure. In *International Conference on Tools and Methods for Program Analysis*, pages 114–126. Springer, 2017.

[120] David Monniaux. A survey of satisfiability modulo theory. In *International Workshop on Computer Algebra in Scientific Computing*, pages 401–425. Springer, 2016.

[121] Tukaram Muske. Improving review of clustered-code analysis warnings. In *International Conference on Software Maintenance and Evolution*, pages 569–572. IEEE, 2014.

[122] Tukaram Muske. *Postprocessing of Static Analysis Alarms*. PhD Dissertation, Eindhoven University of Technology, Eindhoven, The Netherlands, 2020.

[123] Tukaram Muske, Ankit Baid, and Tushar Sanas. Review efforts reduction by partitioning of static analysis warnings. In *International Working Conference on Source Code Analysis and Manipulation*, pages 106–115. IEEE, 2013.

[124] Tukaram Muske and Prasad Bokil. On implementational variations in static analysis tools. In *International Conference on Software Analysis, Evolution, and Reengineering*, pages 512–515. IEEE, 2015.

[125] Tukaram Muske, Advaita Datar, Mayur Khanzode, and Kumar Madhukar. Efficient elimination of false positives using bounded model checking. In *International Conference on Advances in System Testing and Validation Lifecycle*, pages 13–20. IARIA XPS Press, 2013.

[126] Tukaram Muske and Uday P. Khedker. Efficient elimination of false positives using static analysis. In *International Symposium on Software Reliability Engineering*, pages 270–280. IEEE, 2015.

[127] Tukaram Muske and Uday P. Khedker. Cause points analysis for effective handling of alarms. In *International Symposium on Software Reliability Engineering*, pages 173–184. IEEE, 2016.

[128] Tukaram Muske and Alexander Serebrenik. Survey of approaches for handling static analysis alarms. In *International Working Conference on Source Code Analysis and Manipulation*, pages 157–166. IEEE, 2016.

[129] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. Repositioning of static analysis alarms. In *International Symposium on Software Testing and Analysis*, pages 187–197. ACM, 2018.

[130] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. Reducing static analysis alarms based on non-impacting control dependencies. In *Asian Symposium on Programming Languages and Systems*, pages 115–135. Springer, 2019.

[131] Thu-Trang Nguyen, Toshiaki Aoki, Takashi Tomita, and Iori Yamada. Multiple program analysis techniques enable precise check for SEI CERT C coding standard. In *Asia-Pacific Software Engineering Conference*, pages 70–77. IEEE, 2019.

[132] Thu Trang Nguyen, Pattaravut Maleehuan, Toshiaki Aoki, Takashi Tomita, and Iori Yamada. Reducing false positives of static analysis for SEI CERT C coding standard. In *Joint International Workshop on Conducting Empirical Studies*

in Industry and International Workshop on Software Engineering Research and Industrial Practice, pages 41–48. IEEE, 2019.

[133] Lisa Nguyen Quang Do and Eric Bodden. Gamifying static analysis. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 714–718. ACM, 2018.

[134] Paulo Nunes, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios. *Computing*, 101(2):161–185, 2019.

[135] Hideto Ogasawara, Minoru Aizawa, and Atsushi Yamada. Experiences with program static analysis. In *International Software Metrics Symposium*, pages 109–112. IEEE, 1998.

[136] Jan-Peter Ostberg and Stefan Wagner. At ease with your warnings: The principles of the salutogenesis model applied to automatic static analysis. In *International Conference on Software Analysis, Evolution, and Reengineering*, pages 629–633. IEEE, 2016.

[137] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan. Auditing buffer overflow vulnerabilities using hybrid static–dynamic analysis. *IET Software*, 10:54–61, 2016.

[138] Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *Symposium on Software Visualization*, pages 77–86. ACM, 2008.

[139] Riyad Parvez, Paul A. S. Ward, and Vijay Ganesh. Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. In *International Conference on Computer Science and Software Engineering*, pages 116–127. IBM Corp., 2016.

[140] Khoo Yit Phang, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. Triaging checklists: a substitute for a PhD in static analysis. *Evaluation and Usability of Programming Languages and Tools*, 2009, 2009.

[141] Marco Pistoia, Omer Tripp, and David Lubensky. Combining static code analysis and machine learning for automatic detection of security vulnerabilities in mobile apps. In *Mobile Application Development, Usability, and Security*, pages 68–94. IGI Global, 2017.

[142] Andreas Podelski, Martin Schäf, and Thomas Wies. Classifying bugs with interpolants. In *International Conference on Tests and Proofs*, pages 151–168. Springer, 2016.

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

[144] Louis-Philippe Querel and Peter C. Rigby. WarningsGuru: integrating statistical bug models with static analysis to provide timely and specific bug warnings. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 892–895. ACM, 2018.

[145] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. User-guided program reasoning using Bayesian inference. In *Conference on Programming Language Design and Implementation*, pages 722–735. ACM, 2018.

[146] Athos Ribeiro, Paulo Meirelles, Nelson Lago, and Fabio Kon. Ranking source code static analysis warnings for continuous monitoring of FLOSS repositories. In *IFIP International Conference on Open Source Systems*, pages 90–101. Springer, 2018.

[147] Athos Ribeiro, Paulo Meirelles, Nelson Lago, and Fabio Kon. Ranking warnings from multiple source code static analyzers via ensemble learning. In *International Symposium on Open Collaboration*, pages 1–10. ACM, 2019.

[148] Xavier Rival. Abstract dependences for alarm diagnosis. In *Asian Symposium on Programming Languages and Systems*, pages 347–363. Springer, 2005.

[149] Xavier Rival. Understanding the origin of alarms in ASTRÉE. In *International Static Analysis Symposium*, pages 303–319. Springer, 2005.

[150] Neha Rungta and Eric G. Mercer. A meta heuristic for effectively detecting concurrency errors. In *Haifa Verification Conference*, pages 23–37. Springer, 2008.

[151] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for java. In *International Symposium on Software Reliability Engineering*, pages 245–256. IEEE, 2004.

[152] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *International conference on Software engineering*, pages 341–350. ACM, 2008.

[153] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering*, pages 598–608. IEEE, 2015.

[154] Lwin Khin Shar and Hee Beng Kuan Tan. Mining input sanitization patterns for predicting sql injection and cross site scripting vulnerabilities. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1293–1296. IEEE, 2012.

[155] Haihao Shen, Jianhong Fang, and Jianjun Zhao. EFindbugs: Effective error ranking for FindBugs. In *International Conference on Software Testing, Verification and Validation*, pages 299–308. IEEE, 2011.

[156] Mark S. Sherriff, Sarah Heckman, J. Michael Lake, and Laurie Williams. Using groupings of static analysis alerts to identify files likely to contain field failures. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering: Companion Papers*, pages 565–568. ACM, 2007.

[157] Hasan Sözer. Integrated static code analysis and runtime verification. *Software: Practice and Experience*, 45(10):1359–1373, 2015.

[158] Jaime Spacco, David Hovemeyer, and William Pugh. Tracking defect warnings across versions. In *International workshop on Mining software repositories*, pages 133–136. ACM, 2006.

[159] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. In *International Conference on Software Engineering*, pages 120–131. ACM, 2016.

[160] David Svoboda, Lori Flynn, and Will Snavely. Static analysis alert audits: Lexicon & rules. In *Cybersecurity Development*, pages 37–44. IEEE, 2016.

[161] TCS Embedded Code Analyzer (TCS ECA). https://www.tcs.com/tcs-embedded-code-analyzer. [Online: accessed 01-March-2020].

[162] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *International Conference on Automated Software Engineering*, pages 50–59. IEEE, 2012.

[163] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[164] Vassil Todorov, Frédéric Boulanger, and Safouan Taha. Formal verification of automotive embedded software. In *Conference on Formal Methods in Software Engineering*, pages 84–87. ACM, 2018.

[165] Aaron Tomb, Guillaume Brat, and Willem Visser. Variably interprocedural program analysis for runtime error detection. In *International Symposium on Software Testing and Analysis*, pages 97–107. ACM, 2007.

[166] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. ALETHEIA: Improving the usability of static security analysis. In *Conference on Computer and Communications Security*, pages 762–774. ACM, 2014.

[167] Manuel Valdiviezo, Cristina Cifuentes, and Padmanabhan Krishnan. A method for scalable and precise bug finding using program analysis and model checking. In *Asian Symposium on Programming Languages and Systems*, pages 196–215. Springer, 2014.

[168] Radhika D. Venkatasubramanyam and Shrinath Gupta. An automated approach to detect violations with high confidence in incremental code using a learning system. In *International Conference on Software Engineering (Companion Proceedings)*, pages 472–475. ACM, 2014.

[169] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 257–267. IEEE, 2000.

[170] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214, 2007.

[171] David A Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, volume 20, page 0, 2000.

[172] Han Wang, Min Zhou, Xi Cheng, Guang Chen, and Ming Gu. Which defect should be fixed first? semantic prioritization of static analysis report. In *International Conference on Software Analysis, Testing, and Evolution*, pages 3–19. Springer, 2018.

[173] Lei Wang, Qiang Zhang, and PengChao Zhao. Automated detection of code vulnerabilities based on program analysis and model checking. In *International Working Conference on Source Code Analysis and Manipulation*, pages 165–173. IEEE, 2008.

[174] Lili Wei, Yepang Liu, and Shing-Chi Cheung. OASIS: Prioritizing static analysis warnings for Android apps based on app user reviews. In *Joint Meeting on Foundations of Software Engineering*, pages 672–682. ACM, 2017.

[175] Chadd C. Williams and Jeffrey K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, 2005.

[176] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *International Conference on Evaluation and Assessment in Software Engineering*, pages 38:1–38:10. ACM, 2014.

[177] Mingjie Xu, Shengnan Li, Lili Xu, Feng Li, Wei Huo, Jing Ma, Xinhua Li, and Qingjia Huang. A light-weight and accurate method of static integer-overflow-to-buffer-overflow vulnerability detection. In *International Conference on Information Security and Cryptology*, pages 404–423. Springer, 2018.

[178] Zhenbo Xu, Jian Zhang, Zhongxing Xu, and Jiteng Wang. Canalyze: a static bug-finding tool for c programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 425–428, 2014.

[179] Jiangtao Xue, Xinjun Mao, Yao Lu, Yue Yu, and Shangwen Wang. History-driven fix for code quality issues. *IEEE Access*, 7:111637–111648, 2019.

[180] Achilleas Xypolytos, Haiyun Xu, Barbara Vieira, and Amr M. T. Ali-Eldin. A framework for combining and ranking static analysis tool findings based on tool performance statistics. In *International Conference on Software Quality, Reliability and Security Companion*, pages 595–596. IEEE, 2017.

[181] Zhao Hong Yang, Yun Zhan Gong, Qing Xiao, and Ya Wen Wang. Dts-a software defects testing system. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 269–270. IEEE, 2008.

[182] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. An empirical study on classification methods for alarms from a bug-finding static C analyzer. *Information Processing Letters*, 102(2-3):118–123, 2007.

[183] Jongwon Yoon, Minsik Jin, and Yungbum Jung. Reducing false alarms from an industrial-strength static analyzer by SVM. In *Asia-Pacific Software Engineering Conference*, volume 2, pages 3–6. IEEE, 2014.

[184] Lian Yu, Jun Zhou, Yue Yi, Jianchu Fan, and Qianxiang Wang. A hybrid approach to detecting security defects in programs. In *International Conference on Quality Software*, pages 1–10. IEEE, 2009.

[185] Ulas Yüksel and Hasan Sözer. Automated classification of static code analysis alerts: a case study. In *International Conference on Software Maintenance*, pages 532–535. IEEE, 2013.

[186] Bin Zhang, Chao Feng, Bo Wu, and Chaojing Tang. Detecting integer overflow in windows binary executables based on symbolic execution. In *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 385–390. IEEE, 2016.

[187] Dalin Zhang, Dahai Jin, Yunzhan Gong, and Hailong Zhang. Diagnosis-oriented alarm correlations. In *Asia-Pacific Software Engineering Conference*, pages 172–179. IEEE, 2013.

[188] Dalin Zhang, Dahai Jin, Ying Xing, Hailong Zhang, and Yunzhan Gong. Automatically mining similar warnings and warning combinations. In *International Conference on Fuzzy Systems and Knowledge Discovery*, pages 783–788. IEEE, 2013.

[189] Danfeng Zhang and Andrew C. Myers. Toward general diagnosis of static errors. In *Symposium on Principles of Programming Languages*, pages 569–581. ACM, 2014.

[190] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. Effective interactive resolution of static analysis alarms. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

[191] Yuwei Zhang, Ying Xing, Yunzhan Gong, Dahai Jin, Honghui Li, and Feng Liu. A variable-level automated defect identification model based on machine learning. *Soft Computing*, 24(2):1045–1061, 2020.

[192] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006.

[193] Honglei Zhu, Dahai Jin, and Yunzhan Gong. Inter-procedural diagnosis path generation for automatic confirmation of program suspected faults. *Tehnički vjesnik*, 26(3):762–770, 2019.

APPENDIX

Table 4. Summary of data extracted from the relevant papers collected through the initial and extended literature searches. Column *Categories* presents the (sub)-category of the approach identified for the papers listed in *Relevant paper (Other sub-categories)* column. In case a paper proposes multiple alarms postprocessing approaches belonging to two or more sub-categories, the sub-categories other than the main sub-category identified for the paper are presented in column *Relevant paper (Other sub-categories)*. Column *Srch* presents the literature search type that identified the relevant paper: I_K and I_S respectively denote the keywords-based search and snowballing of the initial literature search, and E_K and E_S respectively to denote the keywords-based search and snowballing of the extended literature search. The publication year of the paper is given in column *Year*. Column *Tools used in evaluation* presents static analysis tools used to generate alarms required in evaluation of the proposed approaches. Column *Lang.* presents programming languages supported by the tools listed in column *Tools used in evaluation*, i.e., the programming languages of programs that are analyzed to generate alarms used in the evaluations of the approaches. Columns *Techniques* and *Artifacts used* respectively present techniques and artifacts used to implement the proposed approaches. In column *Tools used in evaluation*, a footnote/biblographic reference is added for a tool only if the tool is referenced for the first time in the table. All the urls in the footnote have been last accessed on August 27, 2021.

Cate-gories	Sr. No.	Relevant paper (Other sub-categories)	Srch	Year	Lang.	Tools used in evaluation	Techniques	Artifacts used
A. Clustering of alarms								
Sound	1	Jiao et al. [71]	E_K	2017	C	DTSC [71]	feature functions	defect model
	2	Lee et al. [99]	E_K	2017	C	SPARROW[7]	abstract interpretation	-
	3	Lee et al. [100]	I_K	2012	C	Airac[8]	abstract interpretation, trace partitioning	-
	4	Muske et al. [123]	I_K	2013	C	TECA[9]	data flow analysis	-
	5	Muske et al. [129]	E_K	2018	C	TCS ECA[10]	data flow analysis	-
	6	Muske et al. [130]	E_K	2019	C	TCS ECA	data flow analysis	-
	7	Zhang et al. [187]	I_K	2013	C	DTSGCC [187]	semantic slicing, error state slicing	-
Unsound	8	Fry et al. [52]	I_K	2013	C, Java	Coverity[11], FindBugs[12]	graph theory	syntactic and structural info
	9	Le and Soffa [97]	I_K	2010	C	Phoenix[13]	fault correlation graphs (graph theory)	modeled error states
	10	Podelski et al. [142]	I_K	2016	Java	Bucketeer [142]	Craig interpolation	semantics-based signatures

Continued on next page

[7] https://github.com/ropas/sparrow

[8] http://ropas.snu.ac.kr/2005/airac5/

[9] https://www.tcs.com/tcs-embedded-code-analyzer

[10] https://www.tcs.com/tcs-embedded-code-analyzer

[11] https://scan.coverity.com/

[12] http://findbugs.sourceforge.net/

[13] https://archive.org/details/phoenixsdkjune2008rc1

Table 4 – continued from previous page

Cate-gories	Sr. No.	Relevant paper (Other sub-categories)	Srch	Year	Lang.	Tools used in evaluation	Techniques	Artifacts used
	11	Sherriff et al. [156]	I_K	2007	C, C++	Matlab[14]	singular value decomposition	alarm signatures
	12	Zhang et al. [188]	I_K	2013	C	DTSGCC	data mining	execution traces
B. Ranking of alarms								
Statistical analysis	13	Jung et al. [76]	I_K	2005	C	Airac [76]	statistical analysis (Bayesian networks)	syntactic alarm contexts
	14	Kremenek and Engler [91]	I_S	2003	C	MC [46]	statistical analysis (hypothesis testing)	number of alarms
History-aware	15	Aman et al. [4]	E_S	2020	Java	PMD[15]	survival analysis	history of alarms
	16	Burhandenny et al. [22]	E_S	2017	Java	PMD	authorship of source files	history of alarms
	17	Kim and Ernst [81]	I_K	2007	Java	FindBugs, PMD, Jlint[16]	statistical analysis	source code repository metrics
	18	Kim and Ernst [82]	I_K	2007	Java	FindBugs, PMD, Jlint	statistical analysis	source code repository metrics
	19	Liu et al. [108] (Clustering - Unsound Simplification of manual inspection - Automated repair)	E_K	2018	Java	FindBugs	CNNs	alarm fix patterns
	20	Williams and Hollingsworth [175]	I_S	2005	Java	FindBugs	repository mining	bug repository metrics, alarm fix history
Feed-back -based	21	Heckman [63]	I_K	2007	Java	FindBugs	statistical analysis	alarm types, code locality, alarm fix hitory
	22	Kremenek et al. [90] (Clustering - Unsound)	I_S	2004	C	MC	machine learning	code locality, user-feedback
	23	Raghothaman et al. [145]	E_K	2018	Java	Bingo [145]	Bayesian inference	probabilistic model
	24	Shen et al. [155]	I_K	2011	Java	FindBugs	-	alarm patterns
	25	Wei et al. [174] (Clustering - Unsound)	E_K	2017	Java	Android Lint[17]	NLP techniques	user reviews

Continued on next page

[14] https://www.mathworks.com/products/matlab.html
[15] https://pmd.github.io/
[16] http://artho.com/jlint/.
[17] https://developer.android.com/studio/write/lint.html

Table 4 – continued from previous page

Cate-gories	Sr. No.	Relevant paper (Other sub-categories)	Srch	Year	Lang.	Tools used in evaluation	Techniques	Artifacts used
Multiple tools	26	Flynn et al. [51]	E_K	2018	C, C++, Java, Perl	SCALe[18]	classification models	code base metrics, alarms fix history
	27	Kong et al. [88] (Ranking - feedback-based)	I_S	2007	C	RATS[19], ITS4 [169], FLAW-FINDER[20]	data fusion	alarm metrics
	28	Lu et al. [111]	E_K	2018	C/C++	Cppcheck[21], CBMC[22], Frama-C[23], Clang Static Analyzer[24]	machine learning	defect types, code structures
	29	Meng et al. [117]	I_S	2008	Java	FindBugs, PMD, Jlint	policy prioritization	defect patterns
	30	Nunes et al. [134]	E_K	2019	PHP	Pixy[25], WAP[26], RIPS[27], phpSAFE[28], WeVerca[29]	1-out-of-N strategy	(Non-) Vulnerable LOCs
	31	Ribeiro et al. [146]	E_K	2018	C, C++	Clang static analyzer, Cppcheck, Frama-C	ensemble learning	labeled alarms
	32	Ribeiro et al.[147]	E_S	2019	C, C++	Cppcheck, Frama-C, Clang Static Analyzer	ensemble learning	labeled alarms
	33	Xypolytos et al. [180]	E_K	2017	C	-	-	test suites
Others	34	Blackshear and Lahiri [17]	I_K	2013	C	ACSPEC [17]	semantic reasoning	predicates and specifications

Continued on next page

[18] https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=473847
[19] https://security.web.cern.ch/recommendations/en/codetools/rats.shtml
[20] https://dwheeler.com/flawfinder/
[21] http://cppcheck.sourceforge.net/
[22] https://www.cprover.org/cbmc/
[23] https://frama-c.com/
[24] https://clang-analyzer.llvm.org/
[25] https://github.com/oliverklee/pixy
[26] https://analysis-tools.dev/tool/wap
[27] http://rips-scanner.sourceforge.net/
[28] https://github.com/JoseCarlosFonseca/phpSAFE
[29] https://d3s.mff.cuni.cz/software/weverca/

Table 4 – continued from previous page

Cate-gories	Sr. No.	Relevant paper (Other sub-categories)	Srch	Year	Lang.	Tools used in evaluation	Techniques	Artifacts used
	35	Boogerd and Moonen [18]	I_K	2006	C	Codesurfer[30]	graph theory	code metrics
	36	Nguyen Quang Do et al. [43] (Simplification of manual inspection - Feedback-based)	E_K	2017	Java	CHEETAH[31]	layered analysis	-
	37	Heo et al. [68]	E_S	2019	C	SPARROW	differential Bayesian inference	differential derivation graph, user feedback
	38	Liang et al. [107]	I_K	2012	Java	-	expressive defect pattern specification notation (EDPSN)	defect patterns
C. Pruning of alarms								
ML	39	Alikhashashneh et al. [2]	E_K	2018	C++	-	SVM, KNN, RIPPER Random forests	source code metrics
	40	Hanam et al. [61]	I_K	2014	Java	FindBugs	machine learning	alarm patterns
	41	Heckman and Williams [65]	I_S	2009	Java	-	machine learning	alarm characteristics
	42	Heo et al. [67]	E_K	2017	C	-	One-class SVM	codebase with bugs
	43	Koc et al. [87]	E_K	2017	Java	FindSecBugs[32]	Bayes and LSTM models	code patterns
	44	Lee et al. [98]	E_K	2019	C,C++	-	CNNs	lexical patterns
	45	Meng et al. [118]	E_S	2017	C	-	machine learning	code property graph
	46	Pistoia et al. [141]	E_K	2017	Java	Phoenix, IBM Security AppScan Source[33]	machine learning	syntactic properties of alarms
	47	Tripp et al. [166] (Simplification of manual inspection - Feedback-based)	E_S	2014	Java-Script	-	machine learning	user feedback
	48	Yüksel and Sözer [185]	I_K	2013	C, C++	-	machine learning	alarm and code characteristics
	49	Yoon et al. [183]	I_K	2014	Java	SPARROW	machine learning	structural characteristics

Continued on next page

[30] https://github.com/pomber/code-surfer

[31] https://blogs.uni-paderborn.de/sse/tools/cheetah-just-in-time-analysis/

[32] https://find-sec-bugs.github.io/

[33] www.ndm.net/sast/ibm-appscan-source

Table 4 – continued from previous page

Categories	Sr. No.	Relevant paper (Other sub-categories)	Srch	Year	Lang.	Tools used in evaluation	Techniques	Artifacts used
	50	Zhang et al. [191]	E_K	2019	C	DTS [181]	machine learning	characteristics of variables
Delta alarms	51	Chimdyalwar and Kumar [30]	I_K	2011	C	TECA	impact analysis	assertions
	52	Lahiri et al. [93]	E_S	2013	C	SymDiff[34]	differential analysis	-
	53	Logozzo et al. [109]	I_K	2014	C#	cccheck [47]	relative correctness	necessary/ sufficient conditions
	54	Spacco et al. [158]	I_S	2006	Java	FindBugs	fuzzier matching algorithms	warning signatures
	55	Venkatasubramanyam and Gupta [168] (Pruning - ML)	I_K	2014	C++	-	learning system	alarm and error patterns
Others	56	Ayewah et al. [10]	I_K	2007	Java	FindBugs	patterns identification	alarm types and patterns
	57	Chen et al. [26]	I_K	2013	C	RELAY [170]	thread specialization	code regions
	58	Das et al. [38]	I_K	2015	C	Angelic-Verifier [38]	abductive inference	angelic assertions, vocabulary
	59	Joshi et al. [75]	E_S	2012	C	CBUGS [75], POIROT[35]	differential analysis	-
	60	Ruthruff et al. [152]	I_K	2008	Java	FindBugs	statistical models	code characteristics /metrics
	61	Wang et al. [172]	E_K	2018	C	Scan-build[36]	-	fixed defects, critical functions
D. Automated False positives elimination (AFPE)								
Model checking - scalability	62	Chimdyalwar et al. [29]	I_S	2015	C	Polyspace[37], TCS ECA	BMC, loop abstraction	program slices
	63	Darke et al. [37]	I_K	2012	C	TECA, CBMC	BMC, loop abstraction	-
	64	Post et al. [143]	I_K	2008	C	Polyspace, CBMC, SATABS[38]	BMC	-
	65	Rungta and Mercer [150]	I_S	2009	Java	Jlint, JPF[39]	greedy depth first search	-

Continued on next page

[34] https://www.microsoft.com/en-us/research/project/symdiff-differential-program-verifier/
[35] http://research.microsoft.com/en-us/projects/poirot/.
[36] https://clang-analyzer.llvm.org/scan-build.html
[37] https://www.mathworks.com/products/polyspace-code-prover.html
[38] https://www.cprover.org/satabs/
[39] https://github.com/javapathfinder/jpf-core

Table 4 – continued from previous page

Cate-gories	Sr. No.	Relevant paper (Other sub-categories)	Srch	Year	Lang.	Tools used in evaluation	Techniques	Artifacts used
	66	Valdiviezo et al. [167]	I_K	2014	C++	Parfait[40], SPIN[41]	model checking, program slicing	abstract programs, program slices
	67	Yu et al. [184]	I_S	2009	Java	-	fuzzy inference, model checking	code characteristics
Model checking - Effici-ency	68	Chimdyalwar and Darke [28]	E_S	2018	C	multiple tools	-	program slices
	69	Darke et al. [36]	E_S	2017	C	ELABMC [36], CBMC	loop abstraction, bounded model checking	program slices
	70	Muske et al. [125]	I_S	2013	C	TECA, CBMC	model checking	-
	71	Muske and Khedker [126]	I_K	2015	C	TCS ECA, CBMC	model checking, data flow analysis	-
	72	Wang et al. [173]	I_S	2008	C	Code-Auditor [173], BLAST[42]	constraint-based analysis, model checking	constraints, program slices
Sym-bolic exe-cution	73	Arzt et al. [7]	E_S	2015	Java	FlowDroid[43]	symoblic execution	-
	74	Feist et al. [49]	E_S	2016	C	BINSEC/SE[44]	dynamic symbolic execution	weighted slices
	75	Gerasimov [57]	E_K	2018	C	Svace[45], Anxiety [56]	dynamic symbolic execution	-
	76	Gerasimov et al. [59]	E_S	2018	C	Avalanche [69]	dynamic symbolic execution	-
	77	Kim et al. [83]	I_K	2010	C	Raccoon[46], Yices[47]	abstract interpretation, symbolic execution	-
	78	Li et al. [101]	I_S	2013	C, C++	Flawfinder[48], SPLINT[49]	trace analysis, symbolic execution	data flow tree
	79	Parvez et al. [139]	E_S	2016	C	WatSym [139], KLEE[50]	symoblic execution	-

Continued on next page

[40] https://llvm.org/pubs/2008-06-SAW-Parfait.html
[41] http://spinroot.com/spin/whatispin.html
[42] http://goto.ucsd.edu/ rjhala/blast.html
[43] https://github.com/secure-software-engineering/FlowDroid
[44] https://binsec.github.io/
[45] https://www.ispras.ru/en/technologies/svace/
[46] https://github.com/evyatarmeged/Raccoon
[47] https://yices.csl.sri.com/
[48] https://dwheeler.com/flawfinder/
[49] https://splint.org/
[50] https://klee.github.io/

Table 4 – continued from previous page

Categories	Sr. No.	Relevant paper (Other sub-categories)	Srch	Year	Lang.	Tools used in evaluation	Techniques	Artifacts used
	80	Zhang et al. [186]	E_K	2016	binaries	IDA pro[51], KLEE	dynamic symbolic execution	-
SMT/ Deductive verification	81	Gadelha et al. [53]	E_S	2019	C,C++	multiple tools	path-satisfiability analysis	-
	82	Nguyen et al. [131]	E_S	2019	C	Rose-checkers[52], Frama-C/WP, CBMC, Cobra[53]	deductive verification, model checking, pattern matching	-
	83	Nguyen et al. [132]	E_K	2019	C	Rose-checkers, Frama-C/WP, CBMC, Cobra	deductive verification	-
	84	Xu et al. [177]	E_S	2019	C	LAID [177], Boolector[54]	path-satisfiability analysis	-
E. Combination of static and dynamic analyses								
All	85	Aggarwal and Jalote [1]	I_S	2006	C	BOON [171], STOBO [62]	-	-
	86	Chebaro et al. [25]	I_K	2012	C	Frama-C, Path-Crawler[55]	program slicing	-
	87	Chen et al. [27]	I_S	2009	x86 binary	IntFinder [27]	taint analysis	suspect instruction set
	88	Csallner et al. [34]	I_K	2005	Java	CnC [34]	constraint solving	abstract/specific error conditions
	89	Csallner et al. [35]	I_K	2006	Java	DSD-Crasher [35]	dynamic inference, dynamic verification	program invariants, test cases
	90	Ge et al. [55]	I_S	2011	C#	DyTa [55]	dynamic test generation	-
	91	Gerasimov and Kruglov [58]	E_K	2018	C	Avalanche	dynamic analysis	-
	92	Kiss et al. [85]	I_K	2015	C, C++, Java	FLINDER-SCA [85]	white-box fuzzing	-

Continued on next page

[51] https://hex-rays.com/
[52] https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=508017
[53] http://spinroot.com/cobra/manual.html
[54] https://github.com/Boolector/boolector
[55] http://pathcrawler-online.com:8080/

Table 4 – continued from previous page

Cate-gories	Sr. No.	Relevant paper (Other sub-categories)	Srch	Year	Lang.	Tools used in evaluation	Techniques	Artifacts used
	93	Li et al. [102]	I_K	2011	C,C++	Polyflow [102], Coverity, Clockwork[56]	-	data flow graphs
	94	Li et al. [103]	I_S	2014	Java	RFBI[103]	predictive dynamic analysis	programmer objections
	95	Li et al. [105]	I_K	2013	C, C++	HP Fortify[57]	concolic testing	-
	96	Padmanabhuni and Tan [137] (Pruning - ML)	E_K	2016	C	CodeSurfer, WEKA[58]	dynamic analysis, machine learning	code characteristics
	97	Sözer [157]	I_K	2015	Java	FindBugs	Runtime monitoring	-
	98	Tomb et al. [165]	I_S	2007	Java	Check‘nŠ-Crash [165]	symbolic execution	-
F. Simplification of manual inspection								
Semi-automatic diag-nosis	99	Barik et al. [14]	E_K	2016	Java	FIXBUGS[59]	slow fixes	-
	100	Gao et al. [54] (Simplification of manual inspection - Automated repair, AFPE - Symbolic execution)	E_K	2016	C	Fortify[60], KLEE	reachability	-
	101	Rival [148]	I_K	2005	-	Astree[61]	dependence analysis	abstract dependances
	102	Rival [149]	I_S	2005	C	Astree	semantic slicing	alarm contexts
	103	Zhang and Myers [189] (Ranking - Statistical analysis)	I_S	2014	OCaml	-	expressive constraint language	constraints
	104	Zhu et al. [193]	E_K	2019	C	DTS	section-whole path generation strategy	inter-procedural diagnosis paths
Feed-back -based	105	Mangal et al. [114]	I_K	2015	Java	Eugene [114]	probabilistic analysis	user-feedback
	106	Sadowski et al. [153]	I_S	2015	Multi-ple	Tricorder [153]	data-driven ecosystem	user-feedback
Check-lists	107	Ayewah and Pugh [8]	I_K	2009	Java	FindBugs	systematic reviewing	review checklist

Continued on next page

[56] https://www.perforce.com/products/klocwork
[57] https://www.esecforte.com/products/hp-fortify/
[58] https://www.cs.waikato.ac.nz/ml/weka/
[59] http://go.barik.net/fixbugs
[60] https://www.esecforte.com/products/hp-fortify/
[61] https://www.absint.com/astree/index.htm

Table 4 – continued from previous page

Cate-gories	Sr. No.	Relevant paper (Other sub-categories)	Srch	Year	Lang.	Tools used in evaluation	Techniques	Artifacts used
	108	Phang et al. [140]	I_S	2009	-	-	checklists-based review	triaging checklists
alarms-relevant queries	109	Dillig et al. [42]	I_K	2012	C	Compass [42], Mistral[62]	abductive inference	alarm-specific queries
	110	Kim et al. [80]	E_K	2016	Java	Java Path Finder[63], FindBugs	symoblic execution	-
	111	Muske and Khedker [127] (Ranking - Others)	E_K	2016	C	TCS ECA	data flow analysis	alarm root causes
	112	Zhang et al. [190] (Ranking - Others)	E_K	2017	Java	URSA [190]	integer linear programming	alarm root causes
Auto-mated repair	113	Bader et al. [12]	E_S	2019	Java	Infer[64], Error Prone[65]	fix-patterns mining	alarms fix history
	114	Bavishi et al. [15]	E_S	2019	Java	FindBugs, PHOENIX [15]	learning from examples	alarms fix history
	115	Marcilio et al. [115]	E_K	2019	Java	SPONGE-BUGS[66], SonarQube[67], SpotBugs[68]	program transformation	alarms fixing templates
	116	Medeiros et al. [116] (Pruning - ML)	E_S	2014	PHP	WAP, Pixy, PhpMinerII [154]	machine learning	labeled alarms
	117	Xue et al. [179]	E_S	2019	Java	SonarQube	fix-patterns mining	alarms fix history
UI & navi-gation tools	118	Anderson et al. [5]	I_S	2003	C, C++	CodeSurfer	code navigation	program dependence graph
	119	Buckers et al. [21]	E_K	2017	Java	PMD, FindBugs, Checkstyle[69]	user-interactive exploration	treemap code structure
	120	Cousot et al. [33]	I_K	2005	C	Astree	code navigation	-
	121	Jetley et al. [70]	I_K	2008	C, C++	CodeSonar[70]	code navigation	-

Continued on next page

[62] https://www.cs.utexas.edu/ tdillig/mistral/index.html

[63] https://github.com/javapathfinder/jpf-core

[64] https://engineering.fb.com/2015/06/11/developer-tools/open-sourcing-facebook-infer-identify-bugs-before-you-ship/

[65] https://errorprone.info/

[66] https://github.com/dvmarcilio/SpongeBugs

[67] https://www.sonarqube.org/

[68] https://spotbugs.github.io/

[69] https://checkstyle.sourceforge.io/

[70] https://www.grammatech.com/codesonar-cc

Table 4 – continued from previous page

Cate- gories	Sr. No.	Relevant paper (Other sub-categories)	Srch	Year	Lang.	Tools used in evaluation	Techniques	Artifacts used
	122	Phang et al. [79]	I_K	2008	C	Locksmith[71]	user interfaces	program paths
	123	Parnin et al. [138]	I_S	2008	Java	NOSE-PRINTS [138]	code visualization	-
Others	124	Arai et al. [6]	I_S	2014	Java	GBC [6]	gamification	points/scores
	125	Ma et al. [112]	E_S	2019	C	Canalyze [178]	constraint solving	program paths
	126	Menshchikov and Lepikhin [119]	E_K	2018	C,C++	multiple tools	report verbosity and generalization	-
	127	Muske [121] (Clustering - Unsound)	I_K	2014	C	TCS ECA	review scope chains	call graphs
	128	Nguyen Quang Do et al. [133]	E_K	2018	-	-	video gaming principles	-
	129	Ostberg and Wagner [136]	I_S	2016	Java	HaST [136], FindBugs	salutogenesis model	various metrics, developer comments
	130	Querel and Rigby [144] (Pruning - Others)	E_S	2018	Java	JLint, FindBugs	statistical models	code commits

[71] https://github.com/polyvios/locksmith