2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation

Survey of Approaches for Handling Static Analysis Alarms

Tukaram Muske
Tata Research Development and Design Centre
Tata Consultancy Services, Pune, India
t.muske@tcs.com

Alexander Serebrenik
Eindhoven University of Technology
Eindhoven, The Netherlands
a.serebrenik@tue.nl

***Abstract*—Static analysis tools have showcased their importance and usefulness in automated detection of code anomalies and defects. However, the large number of alarms reported and cost incurred in their manual inspections have been the major concerns with the usage of static analysis tools. Existing studies addressing these concerns differ greatly in their approaches to handle the alarms, varying from automatic postprocessing of alarms, supporting the tool-users during manual inspections of the alarms, to designing of light-weight static analysis tools. A comprehensive study of approaches for handling alarms is, however, not found.**

In this paper, we review 79 alarms handling studies collected through a systematic literature search and classify the approaches proposed into seven categories. The literature search is performed by combining the keywords-based database search and snowballing. Our review is intended to provide an overview of various alarms handling approaches, their merits and shortcomings, and different techniques used in their implementations. Our findings include that the categorized alarms handling approaches are complementary and they can be combined together in different ways. The categorized approaches and techniques employed in them can help the designers and developers of static analysis tools to make informed choices.

I. Introduction

Static analysis tools have showcased their importance and usefulness in automated detection of code anomalies and defects at an early software development phase [1]–[6]. However, several studies [7]–[9] report that these tools are underused in practice. The large number of alarms reported by the tools and cost involved in manual inspection of the alarms have been observed to be the major reasons for the underuse [8]–[11]. The alarms are warning messages to the tool-user, communicating runtime errors like division by zero, overflows/underflows, and null pointer dereferences, that have been detected via static analysis [12]. In practice, a high percentage of the alarms are found to be falsely reported [8], [9].

The above problem of alarms and associated cost can be addressed either by improving precision of the analysis or by postprocessing the alarms effectively after they are generated. The former approach—improvement of analysis precision—has been extensively considered in the literature [13]–[15]. However, given that verification problems are undecidable in general, reporting of false alarms by these tools is inevitable [16], [17]. Furthermore, many times, the tools compromise on precision to achieve analysis scalability or improved performance [18]. As a consequence, reporting of alarms by a static analysis tool is certain, their number is large, and they need to be inspected manually by the tool-users to partition them into true errors and false positives. Thus, manual handling of alarms is unavoidable and costly. There exists a large number of studies that address the problem through postprocessing of alarms, supporting the manual inspections of alarms, or designing of light-weight static analysis tools. The approaches presented by these studies to handle the alarms differ greatly, and to the best our knowledge, they have not been studied comprehensively before.

In this paper, we survey various approaches for handling of alarms generated by static analysis tools. We use *handling of alarms* to mean the following:

1) Automatic postprocessing of the alarms (after they are generated) to reduce the manual inspection effort (cost) through similarity/correlation-based clustering of alarms, ranking or classification of alarms, false positives elimination, etc.
2) Supporting manual inspections of alarms.

Note that the above described handling of alarms does not consider reducing the number of alarms by making underlined static analysis more precise. That is, it excludes the option of improving precision of analyses, like value analysis and pointer analysis, implemented in the static analysis tools.

The aim of the paper is to provide an overview of various alarms handling approaches, their merits and shortcomings, and different techniques used to implement the approaches. We aim to achieve the goal by answering the research question: *what are possible approaches for handling the static analysis alarms?*

To answer the research question, we performed a systematic literature search and identified 79 papers dealing with the handling of alarms. The search is performed by conducting a keywords-based database search first, followed by snowballing [19], [20]. The combination was to complement the two search approaches from each other: the results of the former approach provided a start set required in the latter approach, and the latter approach identified the relevant papers which were missed by the former approach.

We reviewed the identified 79 papers and categorized the approaches presented by them into seven categories described in Section IV. Furthermore, merits and shortcomings of each of the categories are discussed, and wherever appropriate, a

978-1-5090-3848-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SCAM.2016.25

category is classified into sub-categories based on techniques used to implement the approaches. Our findings include that the identified alarms handling approaches are complementary and they can be combined together in several ways. The categorized approaches and techniques employed in their implementations can serve as choices to the designers and developers of static analysis tools.

The following are the contributions of the paper.

1) Identification of 79 papers dealing with handling of alarms, through a systematic literature search.
2) Study and classification of different alarms handling approaches into seven categories.

Paper outline: Section II describes the literature search. Section III summarizes data extracted from the relevant studies. Section IV describes the identified categories of approaches for handling of alarms, and Section V summarizes their merits and shortcomings. Section VI presents related work, and in Section VII we present conclusions and discuss future work.

II. Literature Search

This section presents details of our literature search performed to collect papers dealing with handling of alarms. The search is conducted[1] by combining keywords-based database search and snowballing.

A. Keywords-based Database Search

A keywords-based database search (KDS) is conducted, inspired by systematic literature reviews [21], [22], in Google Scholar to collect relevant papers presenting approaches for handling of alarms. The keywords used during the search are shown in Table I and their selection was considering the research question in Section I. The selection of keywords resulted into $84 = 7 \times 4 \times 3$ different search strings. A separate search is made at Google Scholar for each of the search strings, and we examined the first 150 papers from results of every search to check their relevance to handling of alarms. During the process, we checked in total 12600 results[2] for their inclusion or exclusion. A paper is included in the collection of relevant papers if it deals with

- a technique, method, or an approach to process the alarms (group, rank, classify, etc) after they are generated by a static analysis tool;
- a method or tool-support for systematic manual inspection of alarms;
- an approach for reducing the number of alarms reported (without improving the analysis precision).

We excluded a paper from the collection if it deals with

- improving precision of the underlined static analyses like value analysis and pointer analysis, or refinements to the analyses (like [13], [14], [23]);
- fault prediction or error/bug report triaging;
- mining of bugs repositories in the context of software maintenance/evolution

[1]The search is conducted during the period of June 4 to June 14 2016.
[2]The number includes duplicates in the search results.

TABLE I: Keywords used for database search

I	1) elimination, 2) reduction, 3) simplification, 4) ranking, 5) classification, 6) reviewing, 7) inspection
II	1) static analysis, 2) automated code analysis, 3) source code analysis, 4) automated defects detection
III	1) alarm, 2) warning, 3) alert

- study of economics or benefits of usage of static analysis tools (like [3], [7]);
- evaluation, comparison, or benchmarking of precision of various static analysis tools (such as [24]–[26]).

We considered the title, abstract, introduction/motivation, conclusion, and sometimes evaluation in a paper while applying the inclusion/exclusion criteria for the paper. A paper satisfying any one of the above inclusion (resp. exclusion) criteria was considered sufficient to include (resp. exclude) the paper. In the case of a paper satisfying both the inclusion and exclusion criteria, priority was given for its inclusion. An included paper was not required to have handling of alarms as its main topic. We ensured only peer-reviewed papers are included in the final collection of relevant papers. This activity identified 49 relevant papers.

B. Snowballing approach

Snowballing [19], [27] is performed after the KDS due to the following reasons: (a) the keywords used (search strings) in Table I might be incomplete due to several terminologies and their synonyms used in the relevant 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 database searches [19], [27]. By conducting snowballing after the database search, we tried to identify and include as many relevant papers as possible, which were missed by the database search [28].

1) Creation of Start Set: To begin with, the snowballing requires a start set having diversity in the included papers to avoid bias towards any specific class of papers or approaches collected during the snowballing iterations performed later. Further, such a start set reduces the risk of missing a paper from clusters of papers not referring to each other [19]. In our literature search, we created the required start set by including all the relevant papers identified through the earlier KDS. Thus, the start set used to begin the snowballing included 49 papers.

2) Backward and Forward Snowballing: Iterations of the forward and backward snowballing were performed after the start set was created. In the backward snowballing, the papers in the reference list of each included paper are examined to identify new papers to be included. In the forward snowballing, papers citing an included paper are examined to identify new papers to include (citation analysis). We performed the citation analysis using Google Scholar. We used the same inclusion/exclusion criteria used during the earlier KDS, to include/exclude a paper during both the backward and forward snowballing.

Two iterations of the backward and forward snowballing were performed during which around 5800 papers were examined for their inclusion or exclusion. This activity identified 30 new relevant papers which were not identified during the earlier KDS. Thus, the combination of the two search approaches helped each approach to complement the other: the KDS provided a good start set to snowballing to start with, and the snowballing augmented KDS by identifying relevant papers that were missed by the KDS. Thus, our literature search performed by combining the two search approaches identified 79 papers that dealt with handling of alarms.

III. Data Extraction and Discussion

This section presents extraction of data from the relevant papers collected using the literature search, followed by a few observations from the data extracted.

A. Data Extraction

We reviewed each of the relevant papers and extracted the following data: (a) alarms handling approaches presented by the papers; (b) techniques and artifacts used in the approaches; (c) tools used for evaluating the approaches; (d) programming languages for which handling of alarms is evaluated.

We used open tagging [29] to classify the approaches presented by the papers. The tagging was performed by the first author. The papers having similar approaches are grouped together, and a broader level approach describing the approaches in the grouped papers is identified as a category of handling of alarms. When a paper is found to present multiple approaches combined, i.e. having a possibility of belonging to multiple groups of the papers, a more notable approach, mostly suggested by the title of the paper is selected. For example, for the study by Kremenek et al. [30] that presents clustering and ranking of alarms and utilizes user-feedback for ranking purpose, we have identified ranking as its primary approach. Moreover, an attempt was made to classify an identified category into sub-categories depending on the characteristics of the approaches, or techniques and tools used to implement the approaches.

B. Results Discussion

As a result of the above categorization, the following seven categories of approaches for handling of alarms are identified.

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

B) *Ranking:* various characteristics of the alarms and source code, history of bug/alarm fixes and code changes, etc are used to rank or prioritize the alarms.

C) *Pruning:* alarms are classified either as actionable or non-actionable, and the non-actionable alarms are pruned.

D) *False positives elimination (FPE):* alarms are processed using more precise techniques like model checking and symbolic execution to automatically identify false positives from the alarms.

E) *Static and dynamic analysis combination (SDC):* static and dynamic analyses are combined to handle the alarms.

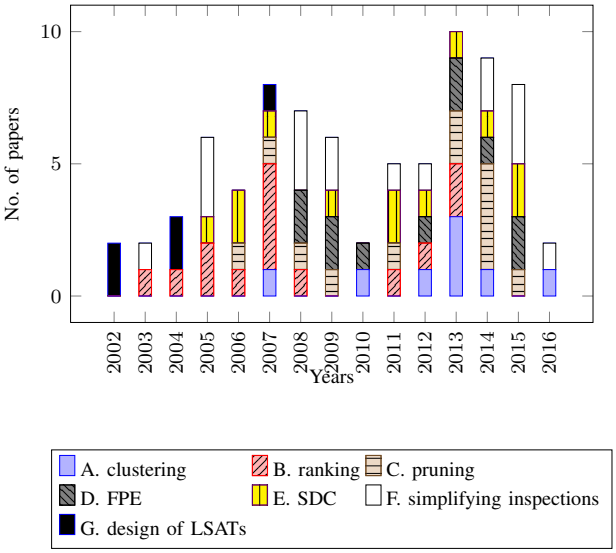

Fig. 1: Number of the relevant papers published year- and category-wise

F) *Simplifying inspections:* manual inspections of alarms are simplified through semi-automatic diagnosis, novel user-interfaces and code navigation tools, review-assisting information, etc.

G) *Design of light-weight static analysis tools (LSATs):* LSATs are designed to counter the problem of large number of analysis alarms.

The distribution of the papers per category is shown year-wise in Figure 1. It indicates that, there is continuous ongoing interest in the topic (handling of alarms), and comparatively a higher number of papers are published recently (in the last three years). Further, simplification of manual inspections has been the more popular category comparatively, while the other four categories—ranking, classification, false positives elimination, and static-dynamic combinations—have received nearly equal popularity.

Figure 2 presents summary of the categorization of approaches for handling of alarms, along with the number of papers in each category. The categorization is described in detail in the next section (Section IV).

Due to the lack of space, we avoid presenting data extracted from each paper. The extracted data is shown in the document available at the following url.

https://sites.google.com/site/scam2016paper/home/download

Following are a few observations made from the data.

- Static analysis tools that analyze C programs are usually used for evaluating the approaches presented.
- Findbugs has been found to be the most popular tool among the different static analysis tools for Java.
- The artifacts used to handle the alarms differ greatly.

IV. Details of Alarms Handling Approaches

This section describes the categories of approaches to handling of alarms, answer to the research question (Section I),

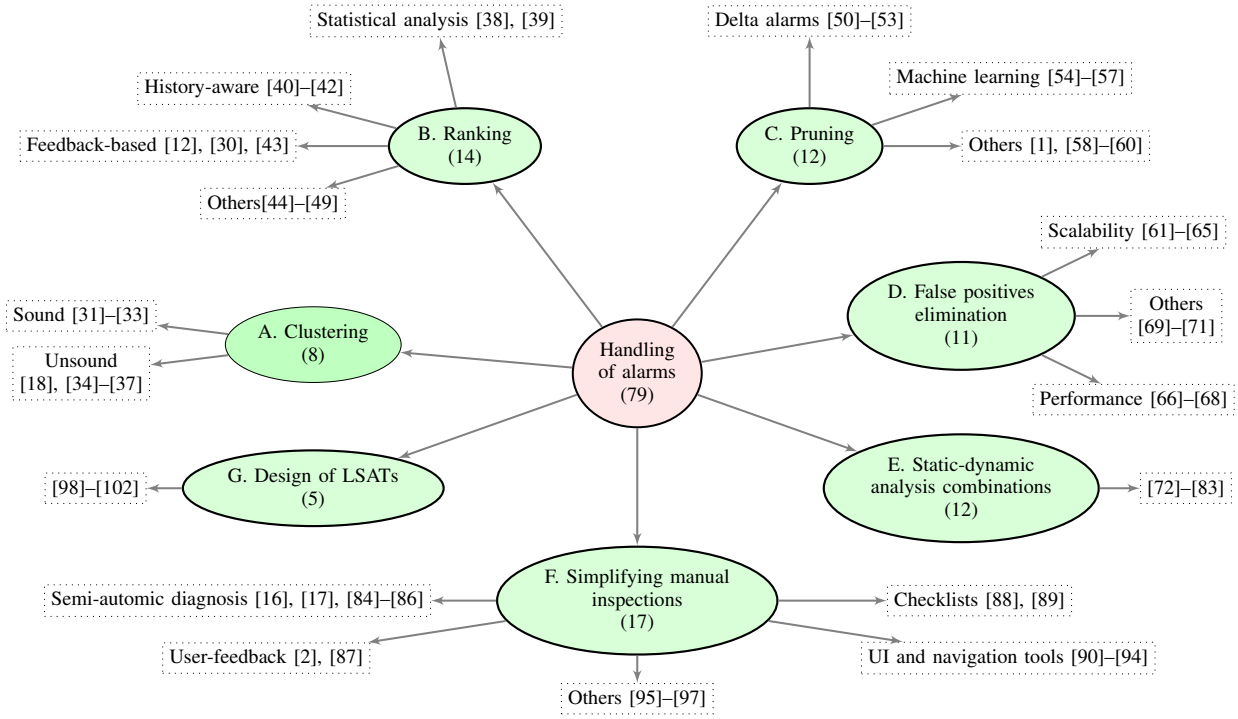

Fig. 2: Summary of various approaches for handling of alarms

along with brief description of a few papers (studies) identified as representatives of their category. Henceforth in the paper, the papers are referred to as *studies*.

A. Similarity/correlation-based Clustering

In this category, static analysis alarms are clustered or partitioned into several groups based on similarity or correlation among the alarms. The alarms in a group being similar/correlated, either only one of them is inspected or all of them are inspected together to eliminate the redundant inspection efforts [31], [35], [36]. The clustering of alarms is further classified as described below.

1) Sound Clustering: The sound clustering is achieved using various analysis techniques that guarantee certain dependencies or relationship among the similar/correlated alarms grouped together. Thus, inspection of only one alarm chosen from a group is sufficient to guarantee inspection of all the alarms from that group. In this sub-category, skipping inspections of alarms other than the chosen alarm does not result in a false negative, thus it is referred to as sound clustering. For example, the studies [31]–[33] form groups of alarms by discovering sound dependencies between the alarms and computing a dominant (leader) alarm for each group. The computation of a dominant alarm is such that if the dominant alarm of a group turns out to be false then all the other alarms belonging to the same group are also false.

The above clustering approach is suitable for any static analysis tool due to the following reasons: (a) there are no false negatives arising due to skipping inspection of alarms other than the dominant alarms; (b) generally their computation cost is also found to be low. Thus, this approach can be implemented by deep analysis tools [50], [103] used to verify safety-critical systems.

2) Unsound Clustering: This sub-category relates to clustering alarms using similarity in syntactic or structural information—code or alarm characteristics—produced by static analysis tools or computed separately. Unlike sound clustering, there are no guarantees on dependencies or relationship among the alarms belonging to the same group, thus this sub-category is referred to as *unsound* clustering. Such clustering is usually based on heuristics and the grouped alarms are inspected together. For example, Fry et al. [36] have used the both structural and syntactic information to partition alarms into groups of related/similar alarms. The partitioning is based on hypothesis that alarms on the same or similar execution paths may be related. On the similar lines, Podelski et al. [35] have proposed a semantics-based signature for an alarm and the signatures are used to group the alarms.

Muske [18] has grouped together the alarms reported for the same program point but belonging to different code-clusters, so that review information is reused to reduce the effort spent in manual inspection of the alarms. Le and Soffa [34] have used cause relationships among the alarms—occurrence of one alarm can cause another alarm to occur—to group the alarms. A correlation graph is constructed by determining the error states of alarms and propagating the effects of the error states along the paths (cause relationships), and it is used 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.

B. Ranking of Alarms

This category corresponds to prioritizing output alarms such that the alarms that are more likely to be true errors are ordered up in the list while the alarms having lower probability to represent an error are pushed to the bottom of the list. This approach is of help when identifying of more defects is demanded in a given time. However, in general, all the ranked alarms needs to be inspected.

1) Statistical Analysis-based Ranking: Statistical analysis has been a common technique to prioritize the alarms. For example, Kremenek and Engler [38] have employed a simple statistical model to rank the 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. [39] 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.

2) History-aware Ranking: In this sub-category, history of alarm fixes has been used as a base to prioritize the alarms. For example, Kim and Ernst [40], [41] have prioritized 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 the similar lines, Williams and Hollingsworth [42] have proposed a ranking scheme based on commonly fixed bugs and information automatically mined from the source code repository.

3) User Feedback-based Self-adaptive Ranking: In this sub-category, user feedback is used to rank the alarms. For example, Shen et al. [43] have first assigned a predefined defect likelihood for each alarm pattern and ranked the alarms based on the the defect likelyhood. Later, the intial ranking is optimized self-adaptively based on the feedback from users. On the similar lines, Kremenek et al. [30] have used user-feedback to dynamically reorder the ranked reports after each inspection. As another example, Heckman [12] has utilized user feedback from analyzed alarms, by combining it with alarm types and code locality, to rank the remaining alarms.

4) Other Techniques: A few other techniques used in ranking of alarms include the following: (a) Static computation of execution likelyhood of the program points at which alarms are reported, also has been used by Boogerd and Moonen [46] for ranking of alarms. (b) Liang et al. [47] have introduced a novel Expressive Defect Pattern Specification Notation (EDPSN) to define a resource-leak defect pattern more precisely. (c) Data mining techniques also have been explored to classify similar alarms [48]. (d) Merging results of multiple static analysis tools that employ different static analysis methods has been used to prioritize the alarms [44], [45]. The merging of results enables results of different tools to validate each other, which in turn, greatly increases or decreases confidence about the false positives and false negatives.

C. Pruning/classification of Alarms

The approaches in this category classify the alarms either as actionable and non-actionable (binary classification). The non-actionable alarms being more likely to be false positives, they are not reported to the users. Thus, this approach can result in false negatives. This category is further classified into the below described sub-categories depending on the techniques or methods employed to achieve the classification.

1) Machine learning-based classification: Several studies, such as [54]–[56], have employed machine learning to differentiate between actionable and non-actionable alarms. For example, Hanam et al. [54] have achieved the classification by finding alarms with similar patterns, where the patterns identification is based on the code surrounding the alarms. Machine learning has been employed to account for semantic and syntactic differences during the patterns identification. Yuksel and Sozer [55] have evaluated 34 machine learning algorithms in their study using 10 different artifact characteristics.

2) Alarm Delta Identification: This sub-category corresponds to applying various analyses, instead of syntactic baselining [51], to identify the alarms that are newly generated as compared to alarms on the previous code version. This is a special case for applications having legacy code components and incremental development (evolving software). It focuses on computing alarm delta between two subsequent versions, to mark which alarms are new, which continue to remain, and which have been disappeared [50]–[53]. For example, Spacco et al. [52] have identified newly generated alarms as compared to the previous version, by matching the alarms through two approaches: *pairing* and *alarm signatures*. Chimdyalwar and Kumar [53] have pruned recurring false positives in evolving large software systems. The pruning is achieved by performing an impact analysis of changes introduced in the current version and suppressing the alarms that are immune to these changes.

Logozzo et al. [51] have introduced a new static analysis technique (Verification Modulo Versions) for reducing 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 uses those conditions to instrument a new version. Later, the instrumented code is verified, which results in pruning of alarms.

3) Other Techniques: A few other techniques include the following: (a) Statistical models are used by Ruthruff et al. [58] to achieve the binary classification of alarms. (b) Das et al. [59] have constrained the analysis verifier to report alarms only when no acceptable environment specification (specified through a vocabulary) exists to prove the assertion. (c) Chen et al. [60] have pruned alarms corresponding to data-races through thread specialization: distinguishing the threads statically by assigning IDs to threads and fixing their number.

D. Elimination of False Positives

In this approach, more precise techniques like model checking and symbolic execution are used to identify and eliminate false positives from alarms generated by static analysis tools.

In a more general approach to do so, an assertion is generated corresponding to every alarm and it is verified using model checkers [61]–[65] or SMT solvers [69], [70]. This approach to handle alarms is more precise and automatic, as compared to the other categories except sound clustering (Section IV-A1), as it eliminates the alarms precisely without any user intervention (inspection). 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 model checking.

1) Achieving Scalability: Post et al. [61] have proposed an incremental approach (context expansion) to achieve scalability during the false positives elimination (FPE). In this approach, the code context for the verification of an assertion is gradually incremented to the callers of the functions, i.e. the verification is started with the minimal context and the context is expanded later on a need basis. This approach has been observed to be beneficial by other studies such as [63], [67] as well.

Further, program slicing is commonly used technique for the state space reduction during the assertion verifications, and in turn, achieving scalability [62]–[64], [66]. A notion of abstract programs also has been proposed by Valdiviezo et al. [62] to achieve scalability of model checkers.

2) Improving Performance/efficiency: The false positives elimination has been found to perform poorly due to the large number of model checking calls. To address this issue, Muske et al. [67], [68] have proposed static analysis-based techniques to predict outcome of a model checking call. The predictions are used to reduce the number of model checking calls to be made and improve on the performance of false positives elimination.

Note that there exists other combinations of static analysis and model checking, like [23], [104], [105], where these two techniques iteratively exchange information between them (tight coupling). We treat this approach different from the approach of false positives elimination through postprocessing of alarms.

E. Static-dynamic analyses combination

As a general theme of this approach, 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 users. This combination requires executing the programs, that is usually absent in the static analysis. A few of these studies adopting this combination approach are described below.

Csallner et al. [72] 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. [73] 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 static-dynamic analysis combinations: confirming/rejecting more number of alarms in a given time [74]. 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. [75] 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. That is, dynamic analysis is *residual* of the static analysis. 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 [75].

F. Simplifying manual inspections of alarms

This category corresponds to simplifying manual inspections of alarms using different approaches like semi-automatic alarm diagnosis, user feedback-based simplifications of inspections, and checklists-based manual inspections.

1) Semi-automatic alarm diagnosis: This sub-category relates to usage of frameworks for semi-automatic error diagnosis or alarm-specific queries for achieving effective and efficient manual inspections of alarms. For example, Dillig et al. [17] have proposed an approach to classify alarms semi-automatically as errors or non-errors 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 error. 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.

Rival [84] has enhanced semantic slicing with information about abstract dependances to help user in alarm inspections by making the inspections more automatic. The abstract dependances relate to chains of dependence among abstract properties likely to capture the cause for an alarm. In another study, Rival [16] has proposed a framework for semi-automatic investigation of alarms. Users are helped to diagnose an alarm by refining an original static analysis into an approximated subset of traces that have actually lead to an alarm. The refinement is achieved through a combination of forward and backward analyses.

2) Simplifying Inspections Using User feedback: Recent studies have been found to capture user-feedback to simplify the manual inspections of alarms. For example, Mangal et al. [87] 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 [87]. 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. [2] have proposed a program analysis platform to build a data-driven ecosystem around static analysis. The platform basically 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 the fixing of alarms reported by the analysis tools.

3) Checklists-based manual inspections: A few studies have been observed to use checklists to systematically guide users during the manual inspections of alarms. Ayewah et al. [88] have proposed use of checklists to enable more detailed review of static analysis alarms. On the similar lines, Phang et al. [89] have used triaging checklists to provide systematic guidance to users during manual inspections of alarms. The users follow the instructions on the checklist, during manual inspections of alarms, to answer each question and to determine conclusions about the alarms. It also proposes 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. Furthermore, the checklists are customized to individual alarms so that a minimum number of questions are answered during triaging of an alarm.

4) 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 tool users while inspecting the alarms manually [91]–[93]. Apart from these tools, several novel approaches (tools) for simplifying the inspections have been proposed. For example, Phang et al. [90] have presented a novel user interface toolkit (path projection) to help users to visualize, navigate, and understand program paths during the inspections. Parnin et al. [94] have used a catalogue of lightweight visualizations to help users in reviewing the alarms. In their study, a simple light-weight visualization is designed for each code smell (alarm). Arai et al. [97] have explored a gamification approach and also proposed a novel gamified tool for motivating developers to remove alarms through manual inspections. The tool proposed calculates scores based on the alarms removed (inspected) by each developer or team.

G. Designing of light-weight static analysis tools (LSATs)

This category deals with systematic designing of static analysis tools to avoid generation of a large number of alarms. Light-weight static analysis tools, different from the deep static analysis tools [50], [103], are designed to effectively detect defects that are commonly made by the developers [98]–[102]. Further, due to the light-weight analysis, LSATs are able to verify very large software systems with improved performance. However, there are no guarantees that all defects of a type will be uncovered.

Hovemeyer and Pugh [98] have implemented automatic detectors for a variety of bug patterns found in Java programs. The resulting tool, Findbugs, has received wider acceptance by the both academia and industry, even though it performs a shallow analysis intraprocedurally. Splint is another tool that uses lightweight static analysis to detect likely vulnerabilities in programs [99]. The analyses performed by Splint being similar to those done by a compiler, they are efficient and scalable while detecting a wide range of implementation flaws. Layman et al. [102] have conducted a study for identifying factors used by developers to decide whether or not to fix an alarm. Based on their findings, they have made several conjectures about the design of static analysis tools so that analysis results are reported effectively. A few of the conjectures made are: (a) alarm descriptions should be as informative and precise as possible, (b) the point at which the analysis tool notifies the developer should be customizable by the developer, (c) the developer must trust that the alarm information is accurate and reliable.

V. Discussion

Table II summarizes the merits and shortcomings of the identified approaches category-wise. It shows that the categories of alarms handling approaches are complementary: shortcoming of one approach are merits of some other approaches and they can complement each other.

We observe that possible combinations of the alarms handling approaches are not widely studied or evaluated. There exists a few studies, such as [30], [68], that consider the combinations of the approaches and find them to be promising. Thus, exploring the possible combinations of these various approaches can be a future research direction to handle the alarms more effectively. For example, only the dominant/leader alarms identified through sound clustering (Section IV-A1) can be processed for their ranking (Section IV-B) or classification/pruning (Section IV-C). In another example, classification of alarms followed by FPE (Section IV-D) can help each other: FPE eliminates false positives from the actionable alarms resulting through the classification, and processing only the actionable alarms (a subset of alarms generated) in the FPE results in improvement of its overall performance.

Furthermore, combinations of the approaches can be implemented with two strategies: sequencing the approaches one after the other (pipelining), and placing 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 classification and FPE 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 true errors. For example, the results obtained in isolation from approaches like ranking, pruning, and static-dynamic analyses combinations, can be merged together to increase confidence about alarms that are more likely to be false positives or errors.

The above combinations of the approaches also may introduce a performance penalty. Thus, performance of such combinations also needs to be studied while studying advantages

TABLE II: Merits and Shortcomings of the approaches

Categories	Merits	Shortcomings
A. Clustering	1. Only dominant alarms need to be inspected (sound clustering). 2. Group-wise inspection of alarms reduces inspection effort.	Unsound clustering may result in false negatives.
B. Ranking	1. Alarms that are more likely to be errors are identified. 2. It does not result in false negatives.	It requires inspecting all the alarms.
C. Pruning/ classification	1. The non-actionable alarms are pruned. 2. Only the actionable alarms are to be inspected.	The pruning may result in false negatives.
D. False positives elimination	It is more precise and automatic as compared to the pruning/classification.	It usually faces issues related to non-scalability and poor performance.
E. Static-dynamic analyses combinations	It presents error scenarios for true alarms.	It requires support for executing the programs (test cases, run-time environment, etc), which is usually absent in static analysis.
F. Simplifying manual inspections	It provides significant aid to the users during manual inspections.	User involvement is a must.
G. Design of LSATs	1. It reduces number of alarms significantly. 2. It is found to be effective in detection of programming errors. 3. It is suitable for non-safety critical systems.	LSATs may not applied for verifying safety-critical systems

and disadvantages of the combinations. We believe, given the high computing power of machines commonly available today, implementing such combinations in practice is feasible and can help practitioners considerably.

Among the identified categories, sound clustering, ranking, elimination of false positives, and simplification of manual inspections, do not result in false negatives[3]. Thus, these approaches can be implemented in static analysis tools that aim at verification of safety-critical systems (false negatives are not allowed during verification of these systems). However, the other approaches—unsound clustering, pruning, and static-dynamic combinations, LSATs—are useful for non-safety critical systems.

VI. Related Work

In this section, we compare our work with a few notable literature reviews and studies about evaluation or benchmarking of tools/techniques dealing with handling of alarms. We start by comparing our study with the systematic literature review conducted by Heckman and Williams [106] as it is a nicely done review of actionable alert identification techniques for static analysis. In this review, approaches presented by 21 different studies that deal with the handling of alarms, are analyzed and they are categorized either as ranking or classification approaches. Among those studies, 10 studies present classification of alarms into actionable or non-actionable, while the other 11 studies dealt with alarms ranking. Compared to this review, our study is more comprehensive as it includes comparatively a large number of papers presenting approaches for handling of alarms. For example, due to selection of additional papers, we could identify the categories like clustering, false positives elimination, and simplification of manual inspections. These categories help to understand various alarms handling approaches better and comprehensively.

In their mapping study, Mendonca et al. [11] have selected and analyzed 51 studies to identify state-of-the-art static analysis techniques and tools, and main approaches developed for handling of alarms. In our study, as our focus was on different approaches through which alarms are handled, we excluded many of the papers that were selected in the mapping study. The excluded papers were dealing with improving analysis precision, study of defects, or evaluation and benchmarking of static analysis tools. Furthermore, our study includes 62 papers that were not included in the mapping study.

The mapping study by Elberzhager et al. [107] has classified and provided analysis of approaches that combine static analysis and dynamic quality assurance techniques. The static quality assurance techniques are dealing with code reviews, inspections, walkthroughs, and usage of static analysis tools, whereas in our survey, combination of static and dynamic analyses is one of the categories focusing only on handling of alarms.

To the best our knowledge, there does not exist other literature surveys or reviews studying the alarms handling approaches. There exists several other studies, like [108]–[110], that evaluate or benchmark various techniques for handling of alarms. Allier et al. [108] have proposed a framework for comparing different alarms ranking techniques and identifying the best approach to rank alarms. The various techniques or algorithms for handling of alarms are compared using a benchmark having programs in Java and Smalltalk languages, and three static analysis tools: FindBugs, PMD, and SmallLint. Using this framework, algorithms for the ranking of alarms are compared. In another study, Liang et al. [109] have proposed an approach for constructing 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/classification of alarms.

VII. Conclusion and Future Work

In this paper, we have studied various approaches to handle the alarms generated by static analysis tools. Our systematic literature search found 79 papers as relevant. We reviewed the papers and classified the approaches presented into seven categories. Also, wherever appropriate, the categories are further classified into sub-categories depending on techniques used. We summarized the merits and shortcomings of these

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

approaches (Section V) to assist designers and developers of static analysis tools to make informed choices. The categorized alarms handling approaches being complementary, they provide an opportunity to combine them. We observe that the identified approaches can be combined together in several ways. Feasibility of the combinations, their advantages and disadvantages, however, need to be studied.

Several of the identified approaches—sound clustering, ranking, elimination of false positives using model checking, simplification of inspections—can help static analysis tools that are used to verify safety-critical systems.

In future, we would be exploring various combinations of the approaches for effective handling of alarms. We believe doing so is feasible and needed, respectively, because of the high computing machines/support commonly available today and the demand from the practitioners in industry to reduce their burden from manual inspection of alarms. Also, we plan to evaluate performance of the identified approaches by means of industrial and open-source case studies.

REFERENCES

[1] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *PASTE 2007*.

[2] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *ICSE 2015*.

[3] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, 2006.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, 2010.

[5] R. Lopes, D. Vicente, and N. Silva, "Static analysis tools, a practical approach for safety-critical software verification," *ESA Special Publication*, vol. 669, 2009.

[6] N. Ayewah and W. Pugh, "The Google FindBugs fixit," in *ISSTA 2010*.

[7] R. Kumar and A. V. Nori, "The economics of static analysis tools," in *ESEC/FSE 2013*.

[8] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *ICSE 2013*.

[9] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *SANER 2016*.

[10] L. Layman, L. Williams, and R. S. Amant, "Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools," in *ESEM 2007*.

[11] V. R. L. de Mendonca, C. L. Rodrigues, F. A. A. de M. N. Soares, and A. M. R. Vincenzi, "Static analysis techniques and tools: A systematic mapping study," in *ICSEA 2013*.

[12] S. S. Heckman, "Adaptively ranking alerts generated from automated static analysis," *Crossroads*, vol. 14, no. 1, 2007.

[13] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani, "Automatically refining abstract interpretations," in *TACAS 2008*.

[14] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, "A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models," in *CGO 2009*.

[15] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella, "Precise widening operators for convex polyhedra," in *SAS 2003*.

[16] X. Rival, "Understanding the origin of alarms in Astrée," in *SAS 2005*.

[17] I. Dillig, T. Dillig, and A. Aiken, "Automated error diagnosis using abductive inference," in *PLDI 2012*.

[18] T. Muske, "Improving review of clustered-code analysis warnings," in *ICSME 2014*.

[19] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *EASE 2014*.

[20] C. Wohlin and R. Prikladnicki, "Editorial: Systematic literature reviews in software engineering," *Information and Software Technology*, vol. 55, no. 6, 2013.

[21] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," *EBSE Technical Report, Ver. 2.3*, no. EBSE-2007-001, 2007.

[22] S. MacDonell, M. Shepperd, B. Kitchenham, and E. Mendes, "How reliable are systematic reviews in empirical software engineering?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, 2010.

[23] M. Junker, R. Huuck, A. Fehnker, and A. Knapp, "SMT-based false positive elimination in static program analysis," in *ICFEM 2012*.

[24] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for Java," in *ISSRE 2004*.

[25] G. Chatzieleftheriou and P. Katsaros, "Test-driving static analysis tools in search of C code vulnerabilities," in *COMPSACW 2011*.

[26] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu, "To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools," in *ASE 2012*.

[27] D. Badampudi, C. Wohlin, and K. Petersen, "Experiences from using snowballing and database searches in systematic literature studies," in *EASE 2015*.

[28] D. Landman, A. Serebrenik, E. Bouwers, and J. 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*, vol. 28, no. 7, 2016.

[29] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: A critical review and guidelines," in *ICSE 2016*.

[30] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation exploitation in error ranking," in *SIGSOFT 2004/FSE-12*.

[31] W. Lee, W. Lee, and K. Yi, "Sound non-statistical clustering of static analysis alarms," in *VMCAI 2012*.

[32] D. Zhang, D. Jin, Y. Gong, and H. Zhang, "Diagnosis-oriented alarm correlations," in *APSEC 2013*.

[33] T. Muske, A. Baid, and T. Sanas, "Review efforts reduction by partitioning of static analysis warnings," in *SCAM 2013*.

[34] W. Le and M. L. Soffa, "Path-based fault correlations," in *FSE 2010*.

[35] A. Podelski, M. Schäf, and T. Wies, "Classifying bugs with interpolants," in *TAP 2016*.

[36] Z. Fry and W. Weimer, "Clustering static analysis defect reports to reduce maintenance costs," in *WCRE 2013*.

[37] M. S. Sherriff, S. S. Heckman, J. M. Lake, and L. A. Williams, "Using groupings of static analysis alerts to identify files likely to contain field failures," in *ESEC-FSE Companion 2007*.

[38] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *SAS 2003*.

[39] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis," in *SAS 2005*.

[40] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in *MSR 2007*.

[41] ——, "Which warnings should I fix first?" in *ESEC-FSE 2007*.

[42] C. C. Williams and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, 2005.

[43] H. Shen, J. Fang, and J. Zhao, "EFindBugs: Effective error ranking for FindBugs," in *ICST 2011*.

[44] D. Kong, Q. Zheng, C. Chen, J. Shuai, and M. Zhu, "ISA: A source code static vulnerability detection system based on data fusion," in *InfoScale 2007*.

[45] N. Meng, Q. Wang, Q. Wu, and H. Mei, "An approach to merge results of multiple static analysis tools," in *QSIC 2008*.

[46] C. Boogerd and L. Moonen, "Prioritizing software inspection results using static profiling," in *SCAM 2006*.

[47] G. Liang, Q. Wu, Q. Wang, and H. Mei, "An effective defect detection and warning prioritization approach for resource leaks," in *COMPSAC 2012*.

[48] D. Zhang, D. Jin, Y. Xing, H. Zhang, and Y. Gong, "Automatically mining similar warnings and warning combinations," in *FSKD 2013*.

[49] S. Blackshear and S. K. Lahiri, "Almost-correct specifications: A modular semantic framework for assigning confidence to warnings," in *PLDI 2013*.

[50] R. D. Venkatasubramanyam and S. Gupta, "An automated approach to detect violations with high confidence in incremental code using a learning system," in *ICSE Companion 2014*.

[51] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear, "Verification modulo versions: Towards usable verification," in *PLDI 2014*.
[52] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking defect warnings across versions," in *MSR 2006*.
[53] B. Chimdyalwar and S. Kumar, "Effective false positive filtering for evolving software," in *ISEC 2011*.
[54] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: Improving actionable alert ranking," in *MSR 2014*.
[55] U. Yüksel and H. Sözer, "Automated classification of static code analysis alerts: A case study," in *ICSM 2013*.
[56] J. Yoon, M. Jin, and Y. Jung, "Reducing false alarms from an industrial-strength static analyzer by SVM," in *APSEC 2014*.
[57] S. Heckman and L. Williams, "A model building process for identifying actionable static analysis alerts," in *ICST 2009*.
[58] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: an experimental approach," in *ICSE 2008*.
[59] A. Das, S. Lahiri, A. Lal, and Y. Li, "Angelic verification: Precise verification modulo unknowns," in *CAV 2015*.
[60] C. Chen, K. Lu, X. Wang, X. Zhou, and L. Fang, "Pruning false positives of static data-race detection via thread specialization," in *APPT 2013*.
[61] H. Post, C. Sinz, A. Kaiser, and T. Gorges, "Reducing false positives by combining abstract interpretation and bounded model checking," in *ASE 2008*.
[62] M. Valdiviezo, C. Cifuentes, and P. Krishnan, "A method for scalable and precise bug finding using program analysis and model checking," in *APLAS 2014*.
[63] P. Darke, M. Khanzode, A. Nair, U. Shrotri, and R. Venkatesh, "Precise analysis of large industry code," in *APSEC 2012*.
[64] B. Chimdyalwar, P. Darke, A. Chavda, S. Vaghani, and A. Chauhan, "Eliminating static analysis false positives using loop abstraction and bounded model checking," in *FM 2015*.
[65] L. Yu, J. Zhou, Y. Yi, J. Fan, and Q. Wang, "A hybrid approach to detecting security defects in programs," in *QSIC 2009*.
[66] L. Wang, Q. Zhang, and P. Zhao, "Automated detection of code vulnerabilities based on program analysis and model checking," in *SCAM 2008*.
[67] T. Muske and U. P. Khedker, "Efficient elimination of false positives using static analysis," in *ISSRE 2015*.
[68] T. Muske, A. Datar, M. Khanzode, and K. Madhukar, "Efficient elimination of false positives using bounded model checking," in *VALID 2013*.
[69] Y. Kim, J. Lee, H. Han, and K.-M. Choe, "Filtering false alarms of buffer overflow analysis using SMT solvers," *Information and Software Technology*, vol. 52, no. 2, 2010.
[70] H. Li, T. Kim, M. Bat-Erdene, and H. Lee, "Software vulnerability detection using backward trace analysis and symbolic execution," in *ARES 2013*.
[71] N. Rungta and E. G. Mercer, "A meta heuristic for effectively detecting concurrency errors," in *HVC 2008*.
[72] C. Csallner and Y. Smaragdakis, "Check 'N' Crash: Combining static checking and testing," in *ICSE 2005*.
[73] C. Csallner, Y. Smaragdakis, and T. Xie, "DSD-Crasher: A hybrid analysis tool for bug finding," in *TOSEM 2008*.
[74] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand, "Program slicing enhances a verification technique combining static and dynamic analysis," in *SAC 2012*.
[75] K. Li, C. Reichenbach, C. Csallner, and Y. Smaragdakis, "Residual investigation: Predictive and precise bug detection," in *ISSTA 2012*.
[76] M. Li, Y. Chen, L. Wang, and G. Xu, "Dynamically validating static memory leak warnings," in *ISSTA 2013*.
[77] B. Kiss, N. Kosmatov, D. Pariente, and A. Puccetti, "Combining static and dynamic analyses for vulnerability detection: Illustration on heartbleed," in *HVC 2015*.
[78] H. Sözer, "Integrated static code analysis and runtime verification," *Software: Practice and Experience*, vol. 45, no. 10, 2015.
[79] J. J. Li, J. Palframan, and J. Landwehr, "Software immunization (SWIM) - a combination of static analysis and automatic testing," in *COMPSAC 2011*.
[80] A. Tomb, G. Brat, and W. Visser, "Variably interprocedural program analysis for runtime error detection," in *ISSTA 2007*.
[81] X. Ge, K. Taneja, T. Xie, and N. Tillmann, "DyTa: dynamic symbolic execution guided with static verification results," in *ICSE 2011*.
[82] P. Chen, H. Han, Y. Wang, X. Shen, X. Yin, B. Mao, and L. Xie, "Intfinder: Automatically detecting integer bugs in x86 binary program," in *ICICS 2009*.
[83] A. Aggarwal and P. Jalote, "Integrating static and dynamic analysis for detecting vulnerabilities," in *COMPSAC 2006*.
[84] X. Rival, "Abstract dependences for alarm diagnosis," in *APLAS 2005*.
[85] D. Zhang and A. C. Myers, "Toward general diagnosis of static errors," in *POPL 2014*.
[86] W. Le and M. L. Soffa, "Generating analyses for detecting faults in path segments," in *ISSTA 2011*.
[87] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *ESEC/FSE 2015*.
[88] N. Ayewah and W. Pugh, "Using checklists to review static analysis warnings," in *DEFECTS 2009*.
[89] K. Y. Phang, J. S. Foster, M. Hicks, and V. Sazawal, "Triaging checklists: a substitute for a phd in static analysis," in *PLATEAU 2009*.
[90] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal, "Path projection for user-centered static analysis tools," in *PASTE 2008*.
[91] P. Anderson, T. Reps, T. Teitelbaum, and M. Zarins, "Tool support for fine-grained software inspection," *IEEE Software*, vol. 20, no. 4, 2003.
[92] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The astreé analyzer," in *ESOP 2005*.
[93] R. P. Jetley, P. L. Jones, and P. Anderson, "Static analysis of medical device software using codesonar," in *SAW 2008*.
[94] C. Parnin, C. Görg, and O. Nnadi, "A catalogue of lightweight visualizations to support code smell inspection," in *SoftVis 2008*.
[95] H. Oumarou, N. Anquetil, A. Etien, S. Ducasse, and K. D. Taiwe, "Identifying the exact fixing actions of static rule violation," in *SANER 2015*.
[96] J. P. Ostberg and S. Wagner, "At ease with your warnings: The principles of the salutogenesis model applied to automatic static analysis," in *SANER 2016*.
[97] S. Arai, K. Sakamoto, H. Washizaki, and Y. Fukazawa, "A gamified tool for motivating developers to remove warnings of bug pattern tools," in *IWESEP 2014*.
[98] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Notices*, vol. 39, no. 12, 2004.
[99] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, vol. 19, no. 1, 2002.
[100] C. C. Williams and J. K. Hollingsworth, "Bug driven bug finders," in *MSR 2004*.
[101] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A system and language for building system-specific, static analyses," in *PLDI 2002*.
[102] L. Layman, L. Williams, and R. S. Amant, "Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools," in *ESEM 2007*.
[103] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic Notes in Theoretical Computer Science*, vol. 217, 2008.
[104] G. Brat and W. Visser, "Combining static analysis and model checking for software analysis," in *ASE 2001*.
[105] A. Fehnker and R. Huuck, "Model checking driven static analysis for the real world: designing and tuning large scale bug detection," *Innovations in Systems and Software Engineering*, vol. 9, no. 1, 2013.
[106] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, no. 4, 2011.
[107] F. Elberzhager, J. Münch, and V. T. N. Nha, "A systematic mapping study on the combination of static and dynamic quality assurance techniques," *Information and Software Technology*, vol. 54, no. 1, 2012.
[108] S. Allier, N. Anquetil, A. Hora, and S. Ducasse, "A framework to compare alert ranking algorithms," in *WCRE 2012*.
[109] G. Liang, L. Wu, Q. Wu, Q. Wang, T. Xie, and H. Mei, "Automatic construction of an effective training set for prioritizing static analysis warnings," in *ASE 2010*.
[110] K. Yi, H. Choi, J. Kim, and Y. Kim, "An empirical study on classification methods for alarms from a bug-finding static C analyzer," *Information Processing Letters*, vol. 102, no. 2-3, 2007.