Information and Software Technology 53 (2011) 363–387

Contents lists available at ScienceDirect

Information and Software Technology

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

A systematic literature review of actionable alert identification techniques for automated static code analysis

Sarah Heckman [*], Laurie Williams

North Carolina State University, 890 Oval Drive, Campus Box 8206, Raleigh, NC 27695-8206, United States

A R T I C L E I N F O

Article history:
Received 2 June 2010
Received in revised form 13 December 2010
Accepted 19 December 2010
Available online 24 December 2010

Keywords:
Automated static analysis
Systematic literature review
Actionable alert identification
Unactionable alert mitigation
Warning prioritization
Actionable alert prediction

A B S T R A C T

Context: Automated static analysis (ASA) identifies potential source code anomalies early in the software development lifecycle that could lead to field failures. Excessive alert generation and a large proportion of unimportant or incorrect alerts (unactionable alerts) may cause developers to reject the use of ASA. Techniques that identify anomalies important enough for developers to fix (actionable alerts) may increase the usefulness of ASA in practice.
Objective: The goal of this work is to synthesize available research results to inform evidence-based selection of actionable alert identification techniques (AAIT).
Method: Relevant studies about AAITs were gathered via a systematic literature review.
Results: We selected 21 peer-reviewed studies of AAITs. The techniques use alert type selection; contextual information; data fusion; graph theory; machine learning; mathematical and statistical models; or dynamic detection to classify and prioritize actionable alerts. All of the AAITs are evaluated via an example with a variety of evaluation metrics.
Conclusion: The selected studies support (with varying strength), the premise that the effective use of ASA is improved by supplementing ASA with an AAIT. Seven of the 21 selected studies reported the precision of the proposed AAITs. The two studies with the highest precision built models using the subject program's history. Precision measures how well a technique identifies true actionable alerts out of all predicted actionable alerts. Precision does not measure the number of actionable alerts missed by an AAIT or how well an AAIT identifies unactionable alerts. Inconsistent use of evaluation metrics, subject programs, and ASAs in the selected studies preclude meta-analysis and prevent the current results from informing evidence-based selection of an AAIT. We propose building on an actionable alert identification benchmark for comparison and evaluation of AAIT from literature on a standard set of subjects and utilizing a common set of evaluation metrics.

Contents

* Corresponding author. Tel.: +1 919 515 2042; fax: +1 919 515 7896.
E-mail addresses: heckman@csc.ncsu.edu (S. Heckman), williams@csc.ncsu.edu (L. Williams).

0950-5849/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2010.12.007

1. Introduction

Static analysis is "the process of evaluating a system or component based on its form, structure, content, or documentation" [26]. Automated static analysis (ASA), like Lint [29], can identify common coding problems early in the development process via a tool that automates the inspection[1] of source code [60]. ASA reports potential source code anomalies,[2] which we call *alerts*, like null pointer dereferences, buffer overflows, and style inconsistencies [25]. Developers inspect each alert to determine if the alert is an indication of an anomaly important enough for the developer to fix. If a developer determines the alert is an important, fixable anomaly, then we call the alert an *actionable alert* [21,22,45]. When an alert is not an indication of an actual code anomaly or the alert is deemed unimportant

[1] An inspection is "a static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems" [28].

[2] An anomaly is a "condition that deviates from expectations, based on requirements specifications, design documents, user documents, or standards, or from someone's perceptions or experiences" [27].

Table 1
Keywords describing static analysis alerts and AAITs.

Alert descriptor	Identification technique
Alert	Classification
Bug	Detection
Defect	Prediction
Fault	Prioritization
Warning	Ranking
	Reduction

to the developer (e.g. the alert indicates a source code anomaly inconsequential to the program's functionality as perceived by the developer), we call the alert an *unactionable alert* [21,22].

Static analysis tools generate many alerts; an alert density of 40 alerts per thousand lines of code (KLOC) has been empirically observed [21]. Developers and researchers found that 35–91% of reported alerts are unactionable [1,4,21,22,31,32,35,36]. A large number of unactionable alerts may lead developers and managers to reject the use of ASA as part of the development process due to the overhead of alert inspection [4,32,35,36]. Suppose, a tool reports 1000 alerts and each alert requires 5 min for inspection. The time to inspect the alerts would take 10.4 uninterrupted 8-h workdays. Identifying the 35–91% unactionable alerts could lead to timesavings of 3.6–9.5 days of developer time. Identification of three or four actionable alerts in two industrial projects programmed in Java was found by Wagner et al. to justify the cost of ASA, if the alerts could lead to field failures [53].

Improving ASA's ability to generate predominantly actionable alerts through development of tools that are both sound[3] and complete[4] is an intractable problem [9,10]. Additionally, the development of algorithms underlying ASA requires a trade-off between the level of analysis and execution time [9]. Methods proposed for improving static analysis include annotations, which could be specified incorrectly and require developer overhead, and allowing the developer to select ASA properties, like alert types, specific to their development environment and project [58].

Another way to increase the number of actionable alerts identified by static analysis is to use the alerts generated by ASA with other information about the software under analysis to prioritize or classify alerts. We call these techniques actionable alert identification techniques[5] (AAIT). Overall, AAITs seek to prioritize or classify alerts generated by ASA *after* an ASA run or use information about the program to guide when ASA is run. AAITs differ from ASA in that they are not refinements to ASA algorithms to approach soundness and/or completeness. Instead, AAITs seek to utilize other information outside of ASA to classify or prioritize ASA alerts quickly. *Classification* AAITs divide alerts into two groups: alerts predicted to be actionable and alerts predicted to be unactionable [21]. *Prioritization* AAITs order alerts by the likelihood an alert is an indication of an actionable alert [21].

The goal of this work is to synthesize available research results to inform evidence-based selection of actionable alert identification techniques. To accomplish this goal we performed a systematic literature review (SLR), which is "a means of evaluating and interpreting all available research relevant to a particular research question or topic area or phenomenon of interest" [33]. The specific objectives of this SLR are the following:

- To identify categories of AAIT input artifacts, both those that come from the alerts generated by ASA and those artifacts that may come from other resources.
- To summarize current research solutions for AAIT.
- To synthesize the current results from AAIT.
- To identify the research challenges and needs in the area of AAIT.

The remainder of this paper is as follows: Section 2 reports the SLR method followed by an overview of the selected studies' characteristics (e.g. publication year and source) in Section 3. Section 4 describes the categories of software artifact characteristics that serve as inputs to AAIT. Section 5 provides a generalized overview of the types of AAITs identified in the selected studies. Section 6 describes the specific studies that use alert classification, while Section 7 describes the specific studies that use alert prioritization. Section 8 provides a combined discussion of all selected studies, and describes a meta- analysis of the results; Section 9 concludes; and Section 10 provides direction for future work from the challenges and needs in the area of AAITs.

2. Overview of systematic literature review method

We used the SLR guidelines described by Kitchenham [33] to develop our SLR protocol. The SLR protocol for addressing the research objectives proposed in Section 1, are presented in the following subsections. The SLR protocol describes the research questions, strategy for searching for related studies, selection of studies for inclusion in the SLR, analysis of the selected studies, and data synthesis, as will be discussed in the following subsections.

2.1. Research questions

We derived our research questions directly from the SLR objectives. We are interested in answering the following research questions:

- *RQ1*: What are the categories of artifacts used as input for AAIT?
- *RQ2*: What are the current approaches for AAITs?
- *RQ3*: What conclusions can we draw about the efficacy of AAITs from results presented in the selected studies?
- *RQ4*: What are the research challenges and needs in the area of AAITs?

Since AAITs are performed after ASA, we first want to understand the sources of information used to generate an alert's classification or prioritization. Next, we want to understand the underlying algorithms for classifying or prioritizing alerts. The third research question seeks to synthesize the results from AAIT and identify how well AAIT techniques work, if at all. The final research question is for the synthesis of open problems in the area of AAITs.

2.2. Search strategy

This section outlines the process for generating search terms, the strategy for searching, the databases searched, and the documentation for the search.

[3] For this research, we consider the generation of an alert as the indication of a potential anomaly, i.e. the goal of ASA tools are to "prove errors exist" [13]. Therefore, sound ASA implies that all reported alerts are (actionable) and unsound ASA implies that unactionable alerts are generated [13].

[4] For this research, we consider the generation of an alert as the indication of a potential anomaly. Therefore, complete static analysis ensures that if there is a place where an anomaly could occur in the source code, the tool reports an alert [13].

[5] AAITs are techniques that identify actionable alerts. Some AAITs have been referred to as false positive or unactionable alert mitigation [21,22], warning prioritization [31,32], and actionable alert prediction [45].

2.2.1. Search terms and strategy

We identified key terms used for our search from prior experience with the subject area. Our main search term is "static analysis" to focus on solutions that identify actionable alerts when using ASA. The other search terms fall into two categories: descriptive names for alerts generated by static analysis and techniques for identification. Table 1 summarizes these terms.

Database search strings combined the keyword "static analysis" with one term from the alert descriptor column and one term from the identification technique column in Table 1 (e.g. "static analysis alert prioritization"). Using each combination of alert descriptor and identification technique, there were 30 search strings for each database. If there was more than one search box with Boolean operators, (e.g. Compendex/Inspec), then "static analysis" was entered in the first box and the alert descriptor and identification technique terms were added to the other boxes with the AND operator selected.

2.2.2. Databases

We gathered the list of potential databases from other SLRs and from the North Carolina State University library's website of suggested databases for Computer Science research. We searched the following databases:

- ACM Digital Library
- Compendex/Inspec
- Computers and Applied Sciences Complete
- ISI Web of Knowledge
- IEEE Xplore
- ScienceDirect
- Springer Link

When databases allow for an advanced search, we excluded non-refereed papers and books. Additionally, if we could restrict papers by subject to Computer Science (e.g. Springer Link) we did so. An initial search was conducted between October 2008 and January 2009. A second search was conducted between October and November of 2010. During the Stage 3 of study selection (as explained in Section 2.3.1), only papers before the end of 2009 were considered.

2.3. Study selection

This section describes the process and documentation used for selecting studies for the SLR of AAITs.

2.3.1. Study selection process

Selection of studies for inclusion in the SLR is a three-stage process: (1) initial selection of studies based upon title; (2) selection of studies based upon reading the abstract; and (3) further selection of studies based upon reading the paper. Table 2 shows the number of papers evaluated at each stage of the selection process. At Stage 1, we started with 17,571 distinct papers (excluding conference proceedings and patents) from the database search and selected 768 that moved to the next stage of papers selection. A 769th paper was added to Stage 2 of paper selection from the verification efforts described in Section 2.3.3. At Stage 3, 70 papers from the original search had relevant abstracts and warranted further reading. The verification efforts added another one paper. The final selected studies consisted of 16 papers. When searching the author's websites for additional work, one of the 16 papers was replaced by a later paper. Four additional papers were added through searching related work of the selected papers and the author's websites.

Inclusion and exclusion criteria were focused on identifying papers that report AAITs that have been used in practice. Specifically, we are interested in AAITs that are used after ASA has been run or to select when and where to run ASA rather than on refinements and enhancements to ASA algorithms. The inclusion and exclusion criteria help refine our definition of AAIT to meet our research objectives. Studies selected at each stage of the selection process met our inclusion criteria:

Table 2
Number of studies evaluated at each stage of selection process.

Stage	Papers	Added papers	Total papers
Stage 1: by title	17,571	0	17,571
Stage 2: by abstract	768	1	769
Stage 3: by paper	70	1	71
Final: selected studies	16	5 (1 replaced)	21

- Full and short peer-reviewed papers with empirical results.
- Post ASA run alert classification or prioritization.
- Focus on automatically identifying if a *single* static analysis alert or a *group* of alerts are actionable or unactionable as opposed to using ASA results to identify fault- or failure-prone files.

Studies rejected at each stage of the selection process met our exclusion criteria:

- Papers unrelated to static analysis or actionable alert identification.
- Theoretical papers about AAITs (i.e. no empirical results).
- Dynamic analyses.
- Hybrid static–dynamic analyses where the static analysis portion of the technique was used to drive improvements to the dynamic portion of the technique rather than using the two techniques synergistically or the dynamic technique to improve the static technique.

During the first two selection stages, we tended to err on the side of inclusion. For the first stage of selection, the inclusion and exclusion criteria were loosened such that any titles that contained the words "static analysis," "model checking," "abstract interpretation," or any variation thereof were selected for inclusion in the next stage of the study, unless they were blatantly outside the scope of the research (e.g. in the field of electrical engineering). Selection of the titles took approximately three days of effort.

In Stage 2, each abstract was read while considering the inclusion and exclusion criteria. The reason for inclusion or exclusion was documented in addition to a paper classification. A classification of "1" denoted papers that should be fully read to determine inclusion. A classification of "2" denoted supporting papers that may be useful for the motivation of the SLR. A classification of "3" denoted papers that did not meet the inclusion criteria. One additional paper was considered at Stage 2, which came from the selection verification as will be discussed in Section 2.3.3. Selection of the abstracts took approximately two days of effort.

Stage 3 incorporated a reading of the selected papers. Details about the quality assessment (Section 2.4.1) and the data extraction (Section 2.4.2) were recorded. A final inclusion or exclusion classification and corresponding reason were recorded. Papers without empirical results were excluded from the SLR at Stage 3. Selection of the papers and recording the appropriate data took approximately a week and a half of effort.

Additional papers were mined from reading the related work sections of selected papers. Three additional papers were identified: [36] from [21,22,31,32,35,45]; [30] from [58]; and [55] from [21,22,32]. Additionally, the websites of the selected papers' authors' of post Stage 3 papers were mined for additional studies that may have been missed during the database search. Because of this search, an additional two studies were identified for inclusion in the final set of 24 (specifically, [6,7]), one of which replaced an earlier study by the same authors (specifically, [5] was replaced by Boogerd and Moonen [7]) due to the reporting of earlier results in addition to new results.

2.3.2. Study selection documentation

Before study selection, duplicate papers identified by different database keyword searches were removed. The study data were stored in Excel and the studies were listed in separate worksheets for each stage of the selection process. For each study, we maintained some or all of the following information:

- Title
- Author(s)
- Conference (abbreviation and full name)
- Publication month
- Publication year
- Abstract

Additionally, for Stages 2 and 3 in the study selection, we included the reason for inclusion and exclusion. After a stage was complete, the selected studies were moved to a new worksheet for the next stage, and any additional information required for selection was obtained.

2.3.3. Selection verification

The first author did the selection of the studies following the process outlined in Section 2.3.1. The second author provided validation of the studies selected at each stage of the selection process, except when verifying the final set of selected studies.

After Stage 1, 292 (1.6%) of the studies were randomly selected for the second author to evaluate. The first author prepared the selection, ensuring that the sample had approximately the same proportion of selected and rejected studies as the full population. Two hundred and seventy-eight (95.2%) of the studies had the same selection by both the first and second author. The second author selected one study the first author did not, and this study was included in Stage 2 of the selection process. The remaining differences were between studies the first author selected but the second author did not. The discrepancy comes because of differences between the interpretations of the hybrid exclusion criteria. The original hybrid exclusion criteria stated, "Hybrid static–dynamic analyses where the static analysis portion of the technique was used to drive improvements to the dynamic portion of the technique." An analysis of the studies that would not have moved on to the second stage of the selection process because of the second author's selection showed that only one of the studies was a selected study in the final set and was a hybrid study. A kappa value [11] of 0.4613[6] (with a 95% confidence interval of 0.1751–0.7475) shows a moderate level of agreement between the two author's selections.

After Stage 2, the second author evaluated the abstracts for 48 (6.3%) randomly selected studies. We again ensured that the sample had approximately same distribution of selected and rejected studies as the full population of studies. Forty-three (89.5%) of the randomly selected studies had the same selection by the two authors. The study the second author classified as "1" was included in Stage 3 of study selection. A kappa value [11] of 0.7022 (with a 95% confidence interval of 0.4552–0.9492) shows a moderate level of agreement between the two author's selections.

2.4. Study analysis

After all stages of the SLR study selection were complete, the next step measured the quality of the selected studies and extracted the data for the SLR from the studies. The following sections describe the data collected from each of the selected studies.

2.4.1. Study quality assessment

We are interested in assessing the quality of each of the selected (e.g. post stage 3) studies. For each study, we answered the questions outlined in Table 3. All of the questions have three possible responses and associated numerical values: yes (1), no (0), or somewhat (0.5). The sum of responses for the quality assessment questions provides a relative measure of study quality. The questions were generated based on the authors experiences with writing and reading research papers and the components that make up a good paper. Therefore, there is potential bias toward the author's selected studies receiving a quality score of 10. However, four additional papers received a quality score of 10 and the average quality score was 8.3. Eleven of the 21 selected studies did not report limitations of their evaluation methodology. Seven of the selected studies did not provide a control or baseline for a comparative evaluation or did not provide limitations for their AAIT.

2.4.2. Study data extraction

For each post Stage 3 selected study (which we will refer to as "selected studies" from this point forward), we extracted the following data:

- Type of reference (journal, conference, workshop)
- Research objective
- AAIT type
- AAIT
- AAIT limitations
- Artifact characteristics
- Evaluation methodology (experiment, case study, etc.)
- Evaluation subjects
- Evaluation metrics and definitions
- Evaluation results
- Static analysis tools used
- Evaluation limitations

The data from the selected studies were maintained in several locations: an internal wiki, paper notes, and Excel spreadsheets. The study's evaluation results were gathered into an Excel spreadsheet by evaluation metrics, subject, and AAIT, which allowed for easier synthesis of common data, to answer RQ3.

2.5. Data synthesis

For each research question defined in Section 2.1, we synthesized the associated data collected from each selected study. Section 4 provides an overview of artifact characteristics, which

Table 3
Quality assessment questions.

Is there a clearly stated research goal related to the identification of actionable alerts?
Is there a defined and repeatable AAIT?
Are the limitations to the AAIT enumerated?
Is there a clear methodology for validating the AAIT?
Are the subject programs selected for validation relevant (e.g. large enough to demonstrate efficacy of the technique) in the context of the study and research goals?
Is there a control technique or process (random, comparison)?
Are the validation metrics relevant (e.g. evaluate the effectiveness of the AAIT) to the research objective?
Were the presented results clear and relevant to the research objective stated in the study?
Are the limitations to the validation technique enumerated?
Is there a listing of contributions from the research?

[6] The kappa values and confidences interval were calculated via the following website: http://faculty.vassar.edu/lowry/kappa.html.

serve as inputs to many of the AAITs and answer research question 1 (RQ1). Section 5 provides a high-level overview of the AAITs, research methodologies, and evaluation metrics used in the selected studies, and provides an overview of the results for RQ2. Section 6 describes the specific results for RQ2 for studies that classify alerts into actionable and unactionable groups. Section 7 describes the specific results for RQ2 for studies that prioritize alerts by the likelihood an alert is actionable. Section 8 summarizes the results for both classification and prioritization techniques for the SLR and answers RQ3. Section 9 concludes and Section 10 discusses future work, and addresses RQ4.

3. Overview of studies

We identified 21 studies in the literature that focus on classifying or prioritizing alerts generated by ASA. An initial look at the studies shows that, with the exception of one study, all work on AAITs occurred during or after 2003 with the most studies (56%) published in 2007 and 2008. Table 4 shows the frequency of publication of AAIT studies by year.

Additionally, we considered the venues of publication for the selected papers. Table 5 shows the publication source for the selected studies and the number of publications from those journals, conferences, or workshops.

Study quality ranged from 3 to 10, where a quality value of 10 is highest, as measured via the 10 questions asked in Section 2.4.1. The quality of each study is presented in Table 6, which lists the selected studies and the identifying information about each. The average study quality was 8.3, which shows that most of the selected studies were of high quality. The selected studies tended to lack a control AAIT for comparison and limitations of the validation methodology. The number of subject programs for each study ranged from 1 to 15, with an average of four subject programs per study.

Table 4
Publication year.

Year	#
1998	1
2003	1
2004	2
2005	2
2006	1
2007	4
2008	6
2009	4
Total	21

4. Software artifact characteristics

One of the things AAITs have in common is that they utilize additional information about software artifacts for the purpose of classifying or prioritizing alerts as actionable or unactionable. The additional information, called *software artifact characteristics*, serves as the independent variables for predicting actionable alerts. We are interested in answering the following question about software artifact characteristics used in AAITs.

- *RQ1*: What are the categories of artifacts used as input for AAIT?

We can generalize the additional information used for AAITs into categories based on the software artifact of origin. The software artifact characteristics are a superset of the four categories summarized below. A deeper discussion of the specific metrics may be found in a supplementary technical report [20] and in the papers selected for the SLR. The artifact characteristics for each of the selected studies may be classified into one of the five categories described below.

- *Alert characteristics (AC)*: Attributes associated with an alert generated via ASA. Alert characteristics are values such as alert type (e.g. null pointer); code location (e.g. package, class, method, line number); and tool-generated alert priority or severity.
- *Code characteristics (CC)*: Attributes associated with the source code surrounding or containing the alert. These attributes may come from additional analysis of the source code or via metrics about the code (e.g. lines per file, cyclomatic complexity).
- *Source code repository metrics (SCR)*: Attributes mined from the source code repository (e.g. code churn, revision history).
- *Bug database metrics (BDB)*: Attributes mined from the bug database. The information from the bug database can be tied to changes in the source code repository to identify fault-fixes.
- *Dynamic analyses metrics (DA)*: Attributes associated with analyzing the code during execution, typically consisting of the dynamic analysis results serving as input to (e.g. invariants to improve static analysis) or a refinement of static analysis (e.g.

Table 5
Publication source.

Publication	Type	#
Asia–Pacific Software Engineering Conference	Conference	1
Computer Software Applications Conference	Conference	1
Empirical Software Engineering and Measurement	Conference	1
Foundations in Software Engineering	Conference	2
Information Processing Letters	Journal	1
International Haifa Verification Conference on Hardware and Software: Verification and Testing	Conference	1
International Conference on Information and Communications Security	Conference	1
International Conference on Quality Software	Conference	1
International Conference on Scalable Information Systems	Conference	1
International Conference on Software Engineering	Conference	1
International Conference on Software Maintenance	Conference	1
International Conference on Software Quality	Conference	1
International Conference on Software Testing, Verification, and Validation	Conference	1
Mining Software Repositories	Workshop	1
Software Metrics Symposium	Symposium	1
Source Code Analysis and Manipulation	Workshop/conference	1
Static Analysis Symposium	Symposium	2
Transactions on Software Engineering and Methodology	Journal	1
Transactions on Software Engineering	Journal	1

Table 6
Summarization of the selected studies.

AAIT name	Study	Study quality	Artifact char. category	ASA	Lang.	Dis. in seconds	AAIT approach	Evaluation methodology
AJ06	Aggarwal and Jalote [1]	6	CC	BOON	C	6.4	Contextual information	Other
ALERTLIFTIME	Kim and Ernst [31]	8.5	AC, SCR	FINDBUGS, PMD, JLINT	Java	7.4	Math/stat models	Other baseline comparison
APM	Heckman and Williams [21]	10	AC	FINDBUGS	Java	7.9	Math/stat models	Benchmark
BM08B	Boogerd and Moonen [6]	9	AC, CC, SCR, BDB	QA C, QMORE	C	6.5	Graph theory	Other
CHECK 'N' CRASH, DSD-CRASHER	Csallner et al. [14]	10	DA	ESC/JAVA	Java	6.6	Dynamic detection	Other model comparison
ELAN, EFAN$_H$, and EFAN$_V$	Boogerd and Moonen [7]	6.5	CC	Unspec.	C	7.8	Graph theory	Other baseline comparison
FEEDBACK-RANK	Kremenek et al. [35]	8	AC	MC	C	7.2	Machine learning	Random and optimal comp.
FIS	Yu et al. [59]	9	CC	Unspec.	Java	6.10	Contextual information	Other
HISTORYAWARE	Williams and Hollingsworth [55]	10	CC, SCR	RETURN VALUE CHECKER	C	6.3	Math/stat models	Other model comparison
HW09	Heckman and Williams [22]	10	AC, CC, SCR	FINDBUGS	Java	6.7	Machine learning	Benchmark
HWP	Kim and Ernst [32]	8	AC, SCR	FINDBUGS, PMD, JLINT	Java	7.5	Math/stat models	Train and test
INTFINDER	Chen et al. [8]	9	DA	Unspec.	C	6.9	Dynamic detection	Other baseline comparison
ISA	Kong et al. [34]	8	AC	RATS, ITS4, FLAWFINDER	C	7.6	Data fusion	Other model comparison
JKS05	Jung et al. [30]	8	CC	AIRIC	C	7.3	Machine learning	Train and test
META-HEURISTIC	Rungta and Mercer [44]	8	CC	JLINT	Java	6.8	Model checking	Random and optimal comp.
MMW08	Meng et al. [38]	3	AC	FINDBUGS, PMD, JLINT	Java	7.10	Data fusion	Other
OAY98	Ogasawara et al. [40]	7	AC	QA C	C	6.1	Alert type selection	Other
RPM08	Ruthruff et al. [45]	10	AC, CC, SCR	FINDBUGS	Java	7.11	Math/stat models	Other model comparison
SCAS	Xiao and Pham [57]	7	CC	Unspec.	C	6.2	Contextual information	Other
YCK07	Yi et al. [58]	9	CC	AIRIC	C	7.7	Machine learning	Train and test
Z-RANKING	Kremenek and Engler [36]	10	CC	MC	C	7.1	Math/stat models	Random and optimal comp.

results from test cases generated to test ASA alerts). Hybrid static–dynamic analyses may help to mitigate the costs associated with each distinct analysis technique [1]. These metrics are associated with concrete executions of the program under analysis and are not appropriate for AAITs that do not incorporate dynamic analyses.

Each of the AAIT described in the selected studies uses software artifact characteristics from one or more of the above categories. The categories used by each AAIT are described in Table 6. Table 7 shows the number of studies that incorporate information from the five categories of artifact characteristics, answering RQ1: alert characteristics, code characteristics, source code repository metrics, bug database metrics, dynamic analyses metrics. A study may have characteristics from more than one category of origin.

The most popular sources of input to AAITs are ACs (48% of the selected studies use ACs) and CCs (57% of the selected studies use CCs). These data show that using information about the alerts, like a developer would when inspecting alerts, may be useful to AAITs for identifying actionable alerts.

Additionally, we can look at the number of studies that incorporate two or more categories of artifact characteristic data. Table 8 shows the number of studies that incorporate data from multiple categories.

Table 7
Studies by category of artifact characteristics.

Artifact characteristics category	Number	Percent
Alert characteristics (AC)	10	48
Code characteristic (CC)	12	57
Source code repository metrics (SCR)	6	29
Bug database metrics (BDB)	1	5
Dynamic analysis metrics (DA)	2	10

5. AAIT approaches and evaluation

The following subsections report the categories of AAIT approaches described in the selected studies as related to RQ2, the evaluation methodology categories, and the subject programs and evaluation methods used for AAIT evaluation. Table 6 presents each of the selected studies and categorizes them with an AAIT type and research methodology. Table 6 also provides a forward reference to the subsection that discusses the specifics of a selected study.

5.1. AAIT approaches

Each selected study describes an AAIT that uses different artifact characteristics, as described in Section 4, to identify actionable

Table 8
Studies with data from multiple categories of artifact characteristics.

Artifact characteristics categories	Number
AC + CC + SCR + BDB	1
AC + CC + SCR	2
AC + SCR	2
CC + SCR	1

and unactionable alerts. Additionally, the approach taken to use the artifact characteristics to predict actionable and unactionable alerts vary by selected study. Analysis of the proposed AAIT identified seven general approaches used by the AAIT in the selected studies, which answer RQ2.

- *RQ2*: What are the current approaches for AAITs?

The following subsections introduce the general AAIT approaches identified from the selected studies. Each of the AAITs described in the 21 selected studies fall into one of the seven approaches listed in Table 9. Table 6 presents the AAIT approach used for each AAIT reported in the selected studies. The most common types of AAIT approaches reported in the literature utilize mathematical and statistical models and machine learning to prioritize and classify alerts.

5.1.1. Alert type selection

ASA tools list the types of problems that can be detected (e.g. a potential null pointer access or an unclosed stream) via a detector or bug pattern, which we call *alert type*. ASA tools may allow for selection of individual alert types by the user. Selecting alert types that are more relevant for a code base leads to the reduction of reported unactionable alerts, but may also lead to the suppression of actionable alerts in the types that were not selected. Alert type selection works best for alert types that tend to be homogeneous by type (e.g. where all alerts of a type are either actionable or unactionable, but not both) [31,32,35]. The alert types that are most relevant may vary by code base. Therefore, a study of the actionability of alert types for a particular code base is required to select the appropriate types. If alerts sharing the same type have been fixed in the past, then the other alerts with that same alert type may be actionable. AAITs that use alert type selection either use the alert history for a project that can be found through mining the source code repository or bug database or additional knowledge about the specific alert types to determine which alert types to select for a project.

5.1.2. Contextual information

Due to imprecision in the analysis, ASA may miss anomalies [1,59]. ASA may also not understand code constructs like pointers, which may lead to a large number of unactionable alerts [1,57]. By understanding the precision of ASA and selecting areas of code that an ASA tool can analyze well (e.g. code with no pointers), the number of generated and unactionable alerts can be reduced. The selection of code areas to analyze by ASA can be a manual or automated process created by knowledge of the ASA tool's limitations.

5.1.3. Data fusion

Data fusion combines data from multiple ASA tools and merges redundant alerts. Similar alerts from multiple tools increase the confidence that an alert is actionable [34].

5.1.4. Graph theory

AAITs that use graph theory to identify actionable alerts take advantage of the source code's structure to provide additional insight into static analysis. System dependence graphs provide both the control and data flow for a program and are used to calculate the execution likelihood for a particular location of code that contains a static analysis alert [4,7]. Other graphs of artifact characteristics, like the source code repository history, can also show the relationship between source code changes that may be associated with openings and closures of static analysis alerts [6].

5.1.5. Machine learning

Machine learning "is the extraction of implicit, previously unknown, and potentially useful information about data" [56]. Machine learning techniques find patterns within sets of data and may then use those patterns to predict if new instances of the data are similar to other instances. AAITs can use machine learning to predict or prioritize alerts as being actionable or unactionable by using information about the alerts and the surrounding code [22,30,35,58].

5.1.6. Mathematical and statistical models

AAITs may use mathematical or statistical models to determine if an alert is actionable or unactionable. In some cases, these AAITs may exploit knowledge about the specific ASA tool to determine if other alerts are actionable or not [36]. Other AAITs may use the history of the code to build a linear model that may predict actionable alerts [31,32,45,55]. Additionally, knowledge about the ASA tools and the observed relationships between the alerts can be used to create mathematical models [21].

5.1.7. Dynamic detection

Unlike static analysis, the results of dynamic analyses do not require inspection because a failing condition is identified through program execution [14]. Dynamic analyses can improve the results generated by static analysis through the generation of test cases that may cause the location identified by the alert to demonstrate faulty behavior [14]. Additionally, by using static analysis to focus automated test case generation, some of the limitations to automated test case generation, like a large number of generated tests, may be reduced [14]. Other dynamic detection techniques utilize instrumented code to identify concretely incorrect executions [8].

Table 9
AAIT approach.

AAIT approach	Number
Alert type selection	1
Contextual information	3
Data fusion	2
Dynamic detection	2
Graph theory	2
Machine learning	4
Mathematical and statistical models	6
Model checking	1

Table 10
Evaluation methodologies for selected studies.

Evaluation methodologies	Number
Baseline comparison	3
Benchmarks	2
Comparison to other AAIT	4
Other	6
Random and optimal comparison	3
Train and test	3

Table 11
Subject program information (uk means unknown, which means the information is not reported in the study).

Name	Version	Static analysis	Lang.	KLOC	# Alerts	Alert density	AAIT
AbsList	uk	JLINT	Java	7.267	6	0.83	META-HEURISTIC [44]
Antiword	uk	n/a	C	27	n/a	n/a	ELAN, EFAN_H, and EFAN_V[7]
Apache Web Server	10/29/2003	RETURN VALUE CHECKER	C	200	738	3.69	HISTORYAWARE [55]
Apache XML Security	1.0.4	ESC/JAVA 2	Java	12.4	111	8.95	CHECK 'N' CRASH, DSD-CRASHER[14]
Apache XML Security	1.0.5 D2	ESC/JAVA 2	Java	12.8	104	8.13	CHECK 'N' CRASH, DSD-CRASHER[14]
Apache XML Security	1.0.71	ESC/JAVA 2	Java	10.3	120	11.65	CHECK 'N' CRASH, DSD-CRASHER[14]
AryList	uk	JLINT	Java	7.169	6	0.84	META-HEURISTIC [44]
Branch 1	uk	SCAS	uk	uk	1692	n/a	SCAS [57]
Branch 2	uk	SCAS	uk	uk	2175	n/a	SCAS [57]
Branch 3	uk	SCAS	uk	uk	3282	n/a	SCAS [57]
Cache	uk	n/a	C	3	n/a	n/a	ELAN, EFAN_H, and EFAN_V[7]
Check	uk	n/a	C	3	n/a	n/a	ELAN, EFAN_H, and EFAN_V [7]
Chktex	uk	n/a	C	4	n/a	n/a	ELAN, EFAN_H, and EFAN_V[7]
Columba	Varies	FINDBUGS, PMD, JLINT	Java	uk	uk	n/a	ALERTLIFETIME [31]
Columba	9/11/2003[a]	FINDBUGS, PMD, JLINT	Java	121	2331[c]	19.26	HWP [32]
Company X	uk	MC	C	uk	uk	n/a	Z-RANKING [36]
Company X	uk	MC	C	800	640	0.80	FEEDBACK-RANK [35]
Cut	uk	n/a	C	1	n/a	n/a	ELAN, EFAN_H, and EFAN_V [7]
cvsobjects	0.5beta	FINDBUGS	Java	1.6	7	4.38	APM [21]
deadlock	uk	JLINT	Java	7.151	6	0.84	META-HEURISTIC [44]
DOM4J	1.6.1	UNNAMED	Java	uk	uk	uk	FIS [59]
Google	uk	FINDBUGS	Java	uk	1652	n/a	RPM08 [45]
Groovy	1.0 beta 1	ESC/JAVA 2	Java	2	34.00	17.00	CHECK 'N' CRASH, DSD-CRASHER[14]
Importscrubber	1.4.3	FINDBUGS	Java	1.7	35	20.59	APM [21]
Indent	uk	n/a	C	26	n/a	n/a	ELAN, EFAN_H, and EFAN_V [7]
iTrust	Fall 2007	FINDBUGS	Java	14.1	110	7.80	APM [21]
Java Path Finder	r1580	UNNAMED	Java	uk	uk	uk	FIS [59]
jbook	1.4	FINDBUGS	Java	1.3	52	40.00	APM [21]
JBoss JMS	4.0 RC 1	ESC/JAVA 2	Java	5	4	0.80	CHECK 'N' CRASH, DSD-CRASHER[14]
jdom	1.1	FINDBUGS	Java	8.4	55	6.55	APM [21]
jdom	revisions up to 1.1	FINDBUGS	Java	9–13.1[b]	420[c]		HW09 [22]
jEdit	Varies	FINDBUGS, PMD, JLINT	Java	uk	uk	n/a	ALERTLIFETIME [31]
Lame	uk	n/a	C	27	n/a	n/a	ELAN, EFAN_H, and EFAN_V [7]
Link	uk	n/a	C	14	n/a	n/a	ELAN, EFAN_H, and EFAN_V [7]
Linux	2.5.8	MC	C	uk	uk	n/a	Z-RANKING [36]
Linux	2.4.1	MC	C	uk	640	n/a	FEEDBACK-RANK [35]
Lucene	8/30/2004[a]	FINDBUGS, PMD, JLINT	Java	37	1513[c]	40.89	HWP [32]
Memwatch	uk	n/a	C	2	n/a	n/a	ELAN, EFAN_H, and EFAN_V [7]
Net-tools	1.46	ISA, RATS, ITS4, FLAWFINDER	C	4.1	uk	n/a	ISA [34]
ngIRCd	0.8.1	UNNAMED	Binary	uk	1	uk	INTFINDER [8]
openssh	2.2.1	UNNAMED	Binary	uk	1	uk	INTFINDER [8]
org.eclipse. core.runtime	3.3.1.1	FINDBUGS	Java	2.8	98	35.00	APM [21]
org.eclipse. core.runtime	Revisions up to 3.3.1.1	FINDBUGS	Java	2–15.5[b]	853[c]	uk	HW09 [22]
P2P Client	uk	UNNAMED	Java	3.129	53	16.9	FIS [59]
php	5.2.5	UNNAMED	Binary	uk	1	uk	INTFINDER [8]
PMD	4.2.4	UNNAMED	Java	uk	uk	uk	FIS [59]
Program A	uk	QA C	C	9	113	12.56	OAY98 [40]
Program B	uk	QA C	C	40	173	4.33	OAY98[40]
Program C	uk	QA C	C	126	1459	11.58	OAY98[40]
Program D	uk	QA C	C	36	1213	33.69	OAY98[40]
Program E	uk	QA C	C	44	1008	22.91	OAY98[40]
Program F	uk	QA C	C	87	2810	32.30	OAY98[40]
Program G	uk	QA C	C	80	779	9.74	OAY98[40]
Pure-ftpd	1.0.17a	ISA, RATS, ITS4, FLAWFINDER	C	25.2	uk	n/a	ISA [34]
python	2.5.2	UNNAMED	Binary	uk	1	uk	INTFINDER [8]
rdesktop	uk	BOON	C	17	uk	uk	AJ06 [1]
Reorder	uk	JLINT	Java	0.044	2	45.45	META-HEURISTIC [44]
Scarab	12/10/2002[a]	FINDBUGS, PMD, JLINT	Java	64	1483[c]	23.17	HWP [32]
slocate	2.7	UNNAMED	Binary	uk	1	uk	INTFINDER [8]
Struts	2.1.6	UNNAMED	Java	uk	uk	uk	FIS [59]
Struts	1.2.7	UNNAMED	Java	uk	uk	uk	FIS [59]

(continued on next page)

Table 11 (*continued*)

Name	Version	Static analysis	Lang.	KLOC	# Alerts	Alert density	AAIT
TVoM	uk	QA C	C	91	9740	107.03	BM08B[6]
TwoStage	uk	JLINT	Java	0.052	2	38.46	META-HEURISTIC [44]
Uni2Ascii	uk	n/a	C	4	n/a	n/a	ELAN, EFAN$_H$, and EFAN$_V$ [7]
Wine	9/14/2004	RETURN VALUE CHECKER	C	uk	2860	n/a	HISTORYAWARE [55]
Wronglock	uk	JLINT	Java	0.038	3	78.95	META-HEURISTIC [44]
wu-ftpd	2.5.0	ISA, RATS, ITS4, FLAWFINDER	C	12.4	uk	n/a	ISA [34]
wzdftpd	uk	BOON	C	uk	uk	n/a	AJ06 [1]
zgv	5.8	UNNAMED	Binary	uk	11	uk	INTFINDER [8]

* Numbers are for build 105, the last build studied.
[a] Date of revision $n/2$.
[b] Size at the latest revision.
[c] Alerts totals across all revisions.

5.1.8. Model checking

Model checking is an analysis technique for checking the correctness of programs by verifying that a model of the program meets certain requirements [24]. Model checking can explore all abstractions of a model and are both sound and complete. However, a limitation of model checkers are the number of states that can be generated when modeling a program. The state explosion problem can make model checking too expensive to use. Directed model checking, like [44], uses ASA alerts to direct model checking to certain areas of the program.

5.2. AAIT evaluation methodology

In the selected studies, all of the alert prioritization techniques were evaluated using an example (as defined in Ref. [46]) on one or more subject programs. Table 10 summarizes the methodologies used to evaluate AAITs in the selected studies. Table 6 identifies which evaluation methodology was used for each AAIT. There is no standard evaluation methodology used to evaluate AAITs in the literature. Each of the evaluation methodologies have their strengths and weaknesses for evaluating AAIT. The remainder of this section discusses the evaluation methodologies used in the selected studies.

5.2.1. Baseline comparison

Eleven of the selected studies used a standard baseline for evaluation (e.g. benchmarks, comparison with other AAITs, random and optimal comparison, or train and test), which are discussed in later sections. Two studies [7,31] used a non-standard baseline for evaluation. The ALERTLIFETIME AAIT prioritized alert types by lifetime, and were evaluated against the ASA tool's ordering of alerts [31]. The ELAN, EFAN$_H$, and EFAN$_V$ AAIT prioritize alerts by their execution likelihood and the prioritizations were compared to the actual execution likelihood as generated by an automated test suite [7]. INTFINDER [8] is evaluated by running the tool on code containing known vulnerabilities.

5.2.2. Benchmarks

Benchmarks provide an experimental baseline for evaluating software engineering theories, represented by techniques (e.g. AAIT), in an objective and repeatable manner [48]. A *benchmark* is defined as "a procedure, problem, or test that can be used to compare systems or components to each other or to a standard" [26]. Benchmarks represent the research problems of interest and solutions of importance in a research area through definition of the motivating comparison, task sample, and evaluation measures [47]. The task sample can contain programs, tests, and other artifacts dependent on the benchmark's motivating comparison. A benchmark controls the task sample reducing result variability, increasing repeatability, and providing a basis for comparison [47]. Additionally, successful benchmarks provide a vehicle for collaboration and commonality in the research community [47].

5.2.3. Comparison to other AAIT

Comparing a proposed AAIT to other AAIT in literature provides another type of baseline. For the best comparisons, the model should exist in the same domain and be run on the same data sets. Model comparison can show the underlying strengths and weaknesses of the models presented in literature and provides additional efficacy about the viability of AAITs.

5.2.4. Random and optimal comparison

Random and optimal orderings of alerts are baselines used to compare AAITs prioritizations [35,36] and may be used in benchmarks (as discussed in Section 5.2.2). The optimal ranking is an ordering of alerts such that all actionable alerts are at the top of the ranking [21,35,36]. A random ranking is the random selection of alerts without replacement [21,35,36]. The random ordering provides a "probabilistic bounded time for an end-user to find a bug, and represents a reasonable strategy in the absence of any information with which to rank reports" [35]. Randomness can be used in other techniques, like the generation of a random depth-first search as a comparison baseline for the META-HEURISTIC [44] technique. Only random and optimal comparisons reported in the selected study were discussed in this SLR.

5.2.5. Train and test

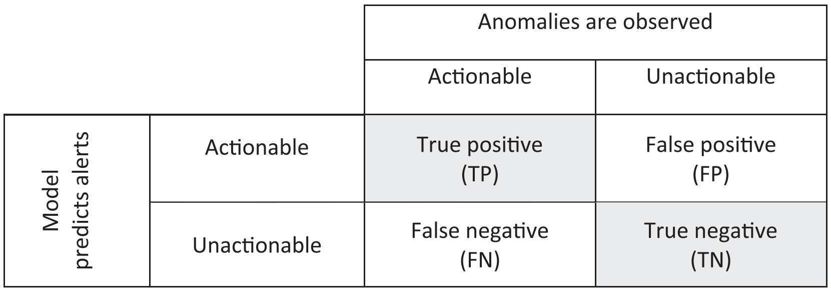
Train and test is a research methodology where some portion of the data are used to generate a model and the remaining portion of the data are used to test the accuracy of the model at predicting or classifying the instances of data [56]. For models that use the project history, the training data may come from the first i revisions of source code [22,32]. The test data then is the remaining $i+1$ to n revisions [22,32]. For models that are generated by considering a set of alerts, the confidence in the accuracy of the models is increased by randomly splitting the data set into train and test sets many (e.g. 100 splits) times [58].

5.2.6. Other

Selected studies that do not use one of the above high-level research methodologies, have been grouped together into the other category. The comparative metrics specific to the studies classified as 'other' are discussed in the AAIT subsections.

5.3. Subject programs

Each of the selected studies provides some level of experimental validation. In Sections 6–8, we compare and consolidate the results presented in the selected studies; however, the synthesis of results is limited by the different domains, programming



		Anomalies are observed	
		Actionable	Unactionable
Model predicts alerts	Actionable	True positive (TP)	False positive (FP)
	Unactionable	False negative (FN)	True negative (TN)

Fig. 1. Classification table (adapted from Zimmerman et al. [61]).

languages, and ASA used in the studies. The key for synthesizing data between selected studies is a common set of subject programs, which implies a common programming language, and ASA(s).

The subject programs used for evaluation of an AAIT varied across the selected studies. Most subject programs are distinct between selected studies. Authors with more than one selected study tended to use the same subject programs across their studies; however, the version of the subject programs varied by study. Table 11 lists the subject programs used in the selected studies and key demographic metrics of the subject programs. Forty-eight distinct subject programs were explicitly used to evaluate the AAITs in the 21 studies; however, 55 subject programs are listed to show how versions differed across AAIT evaluations. Two of the selected studies did not mention the subject programs for AAIT evaluation explicitly [30,38].

Subject program demographic metrics consist of those metrics used for sizing and ASA. Specifically, we are interested in the size of the subject program in terms of KLOC, the number of alerts generated, the ASA run, and the language of the subject programs. These data, if available, are summarized in Table 11. For each of the subject programs with information about size (in KLOC) and the number of alerts, we can measure the alert density for a subject program. However, the alert density varies by ASA tool and the specific ASA configuration the study authors choose to run on their subject programs.

5.4. Metrics for the evaluation of classification AAIT

Alert classification techniques predict whether alerts are actionable or unactionable. We define the key metrics associated with alert classification below:

- *True positive classification (TP)* [6,21,22,40,61]: Classifying an alert as actionable when the alert is actionable.
- *True negative classification (TN)* [21,22,61]: Classifying an alert as unactionable when the alert is unactionable.
- *False positive classification (FP)* [21,22,34,61]: Classifying an alert as actionable when the alert is actually unactionable.
- *False negative classification (FN)* [21,22,34,61]: Classifying an alert as unactionable when the alert is actually actionable.

We are focusing on the classification of alerts identified by the static analysis tool; therefore, we are not considering software anomalies not found by static analysis tools. Fig. 1 is a classification table that model the metrics discussed above.The following metrics evaluate the classification of static analysis alerts:

- *Precision* [1,21,22,32,55,56]: The proportion of correctly classified anomalies (TP) out of all alerts predicted as anomalies (TP + FP). The precision calculation is presented in Eq. (1).

$$\text{Precision} = (TP)/(TP + FP) \tag{1}$$

- *Recall (also called true positive rate or sensitivity)* [21,22,55,56]: The proportion of correctly classified anomalies (TP) out of all possible anomalies (TP + FN). The recall calculation is presented in Eq. (2).

$$\text{Recall} = (TP)/(TP + FN) \tag{2}$$

- *Accuracy* [7,21,22,56]: The proportion of correct classifications (TP + TN) out of all classifications (TP + TN + FP + FN). The accuracy calculation is presented in Eq. (3).

$$\text{Accuracy} = (TP + TN)/(TP + TN + FP + FN) \tag{3}$$

- *False positive rate* [55,56]: The proportion of unactionable alerts that were incorrectly classified as actionable (FP) out of all unactionable alerts (FP + TN). The equation for false positive rate is presented in Eq. (4).

$$\text{False positive rate} = (FP)/(FP + TN) \tag{4}$$

- *Number of test cases generated* [1,14]: The number of automated test cases generated by hybrid techniques that generate test cases. This metric is only appropriate to use with those AAITs that utilize dynamic detection techniques.
- *Error density* [44]: The probability of finding a fault in a program. The probability is "the ratio of the number of [fault] discovering trials over the total number of trials executed for a given model and technique" [44].

5.5. Metrics for the evaluation of prioritization AAIT

Prioritization AAITs can be evaluated using classification metrics (discussed in Section 5.4) if a threshold is specified that divides the alerts into actionable and unactionable sets. Other metrics are also used to evaluate prioritization AAITs. Several correlation techniques compare an AAIT's prioritization of alerts with a baseline prioritization like an optimal ordering of alerts:

- *Spearman rank correlation* [21]: Measuring the distance between the rank of the same alert between two orderings. A correlation close to 1.0 implies that the two orderings are very similar.
- *Wall's unweighted matching method* [54]: Measures how closely the alerts prioritized by the AAITs match the actual program executions.
- *Pearson correlation coefficient, r* [6,31,49]: The correlation coefficient is a measure of the strength of association between independent and dependent variables.
- *Chi-square test* [55]: Comparison of false positive rates to see if the use of an AAIT produces a statistically significant reduction.

- *Area under the curve (AUC)* [21,56,58]: A measure of the area under the graph of the number or percentage of actionable alerts identified over time. The AUC may be measured for many graphs such as the receiver operator characteristic (ROC) curve [56,58] and anomaly detection rate curve [21]. A ROC curve plots the percentage of true positives against the percentage of false positives at each alert inspection. The anomaly detection rate curve plots the number of anomalies or faults detected against the number of alerts inspected.
- *Number of alerts inspected before all actionable alerts identified* [35,36]: The number of alert inspections required to identify all actionable alerts.
- *Prioritization technique's improvement over random* [35,36]: The ratio of the prioritization's AUC and the random ordering's AUC over all or some percentage of possible alerts.

6. Classification AAITs

Classification AAITs divide alerts into two groups: alerts likely to be actionable and alerts likely to be unactionable [21]. The subsections below describe the 10 out of 21 AAITs from the selected studies that are classification AAITs.

For each AAIT, we report the paper describing the AAIT, the input to the AAIT in the form of artifact characteristics used (described in Section 4); the ASA used; programming language the ASA analyzes; AAIT type (described in Section 5.1); and research methodology (described in Section 5.2). If no name is provided for the AAIT in the selected study, we create a name based on the first letter of the first three authors' last names and the last two digits of the year of publication. The AAITs are listed in order of publication year and then the first author's last name.

6.1. OAY98

- Study: Ogasawara et al. [40]
- Artifact characteristics: AC
- ASA: QA C[7]
- Language: C
- AAIT type: Alert type selection (5.1.1)
- Research methodology: Other (5.2.6)

Objective: Ogasawara et al. [40] present a method for using ASA during development whereby only the alert types that identify the most actionable alerts are used.

AAIT method: The static analysis team, using their experiences with ASA, identified 41 key alert types out of 500 possible alert types from the QA C tool. Removing alert types that are not commonly indicative of anomalies reduced the reported unactionable alerts.

Method limitations: One of the unstated limitations of OAY98 is that not all of the alerts of the types removed from the analysis may be unactionable (i.e. actionable alerts were suppressed resulting in a false negative). For the alert types that remain in the analysis, there may still be many unactionable alerts that are generated.

Evaluation methodology: Ogasawara et al. evaluate their technique by investigating the number of alerts and the effectiveness of the alert's messages.

Evaluation subjects: The OAY98 technique was evaluated on seven subject programs, described in Table 11.

[7] QA C is a static analysis tool developed by programming research (http://www.programmingresearch.com/qac_main.html).

Evaluation metrics: The evaluation metrics were the number of alerts, lines of code, alert density, and the number of alerts that were fixed.

Evaluation results: The overall result from Ogasawara et al.'s study is that static analysis is an effective technique for identifying problems in source code. The teams performed code reviews in areas of code containing alerts and found that using static analysis results helped guide code review efforts. Eighty-eight of 250 alerts (35%) were associated with areas of code that were inspected and corrected, implying the alerts were actionable. Thirty percent of those actionable alerts were found to be serious problems in the system.

6.2. SCAS

- Study: Xiao and Pham [57]
- Artifact characteristics: CC
- ASA: UNSPECIFIED
- Language: C
- AAIT type: Contextual information (5.1.2)
- Research methodology: Other (5.2.6)

Objective: Xiao and Pham [57] use contextual information about the code under analysis to extend a static analysis tool.

AAIT method: Unactionable alert reduction was added to three different alert detectors of an unspecified ASA tool: (1) memory leak, (2) missing break, and (3) unreachable code. The memory leak detector keeps track of pointers, especially to global variables, at the function level, and searches for memory leaked specifically by local pointers. If a local pointer does not have a memory leak, then the alert is unactionable and is not reported to the developer. The missing break detector uses belief analysis. Belief analysis uses the source code to infer the developer's beliefs about software requirements. The beliefs inferred from the context of the source code are combined with a lexical analysis of the comments to determine if missing break alerts are actionable or unactionable. The unreachable code detector maintains a database of patterns that suggest unreachable code. Alerts reported about unreachable code may be compared with the patterns in the database. Additionally, any unreachable code alert suppressed by the developer in the user interface of SCAS is transformed into the constituent pattern and recorded in the database.

Method limitations: No limitations were explicitly provided by Xiao and Pham.

Evaluation methodology: SCAS was applied to three branches or projects and the number of generated alerts and the time for analysis were analyzed and compared.

Evaluation subjects: The evaluation subjects were three branches or projects, which are listed in Table 11.

Evaluation metrics: The following evaluation metrics were used: number of files, total number of generated alerts, and number of filtered alerts.

Evaluation results: The SCAS AAIT suppresses 33% of the generated alerts.

6.3. HISTORYAWARE

- Study: Williams and Hollingsworth [55]
- Artifact characteristics: CC, SCR
- ASA: RETURN VALUE CHECKER
- Language: C
- AAIT type: Mathematical and statistical models (5.1.5)
- Research methodology: Other baseline comparison (5.2.4)

Objective: Williams and Hollingsworth [55] use source code repository mining to drive the creation of an ASA tool and to improve the prioritized listing of alerts.

AAIT method: Adding a RETURN VALUE CHECKER on a function call was a common bug fix in the Apache httpd[8] source code repository. Identifying locations where a return value of a function is missing a check is automated via an ASA tool. Alerts associated with the called function are grouped together, and the called functions are ranked using the HISTORYAWARE prioritization technique. HISTORYAWARE first groups functions by mining the software repository for instances where the function call's return value had a check added, closing an alert for an earlier version. Next, the current version of source code is considered by counting the number of times a return value of a function is checked. The functions are prioritized by the count of checked functions. If the return value of a function is checked most of the time, then the prioritization of that function is high, indicating that instances where the return value is not checked are likely actionable. However, if the return value of a function is almost never checked, then the alerts are likely unactionable. When the return value of a called function is always or never checked, the tool does not alert the developer because there are no inconsistencies.

Method limitations: One limitation of the tool is that CVS does not track the versions of the libraries associated with a project, which means that an earlier revision may not build properly without the correct library. Additionally, CVS is limited in identifying moved files, directory renames, and interleaving between users, which reduces the effectiveness of the AAIT.

Evaluation methodology: A case study compares a NAÏVERANKING of the alerts based on the current version of code (e.g. the contemporary context) and the HISTORYAWARE prioritization.

Evaluation subjects: HISTORYAWARE was evaluated using two C programs: Wine[9] and Apache httpd,[10] both of which are listed in Table 11.

Evaluation metrics: The precision and recall of the classifications are measured.

Evaluation results: The precision of the top 50 alerts generated by static analysis is 62.0% for HISTORYAWARE and 53.0% for NAÏVERANKING for Wine and 42.0% for HISTORYAWARE and 32.0% for NAÏVERANKING for Apache httpd. The HISTORYAWARE ranking has a false positive rate between approximately 0% and 70% across the contemporary context of the alerts. The NAÏVERANKING false positive rate is between 50% and 100% on the same contemporary context.

6.4. *AJ06*

- Study: Aggarwal and Jalote [1]
- Artifact characteristics: CC
- ASA: BOON [52]
- Dynamic analysis: STOBO [19]
- Language: C
- AAIT type: Contextual information (5.1.2)
- Research methodology: Other (5.2.6)

Objective: Aggarwal and Jalote [1] identify potential buffer overflow vulnerabilities, as represented by the strcpy library function in C source code, quickly and effectively through a combination of static and dynamic analysis.

AAIT method: The ASA tool, BOON [52], has difficulty in understanding aliased pointers, which may lead to BOON missing buffer overflow vulnerabilities. Dynamic tools, like STOBO [19], can find vulnerabilities where static analysis fails. Dynamic analysis requires the generation of test cases, which can increase the time required to use the tool. AJ06 combine static and dynamic analyses to identify areas of code that require buffer overflow analysis and marks the code where pointers are aliased and where they are not. The former areas of code necessitate dynamic analysis, while buffer overflow vulnerabilities can be found by ASA in the latter code areas.

Method limitations: The AJ06 technique is limited by several of the C language features that are difficult to analyze. Specifically, performance may be impacted by marking strcpy functions. Additionally, structure element aliasing is incorrectly handled.

Evaluation methodology: The study of the AJ06 AAIT does not define a specific research methodology for the experiments; however, we can infer a comparison of the hybrid approach to the performance of the individual static and dynamic approaches. The goal of the analysis is to determine if there is an increase in the accuracy of the static analysis alerts generated and a reduction in test cases (and therefore runtime overhead) for the dynamic analysis. The results of the hybrid analysis are manually audited.

Evaluation subjects: AJ06 is run on subject programs rdesktop[11] and wzdftpd,[12] which are described in Table 11.

Evaluation metrics: The evaluation metrics are accuracy and test case reduction.

Evaluation results: The analysis of wzdftpd showed that only 37.5% of the dangerous strcpy functions in the code required dynamic analysis. The tool for identifying aliased pointers, AJ06, is limited when structured elements are aliased. Additionally, the buffer overflow vulnerability is not restricted to strcpy functions only. Therefore, the technique can only identify a subset of buffer overflow vulnerabilities.

6.5. *BM08B*

- Study: Boogerd and Moonen [6]
- Artifact characteristics: AC, CC, SCR, BDB
- ASA: QA C and custom front end, QMORE
- Language: C
- AAIT type: Graph theory (5.1.3)
- Research methodology: Other (5.2.6)

Objective: Boogerd and Moonen [6] present a technique for evaluating the actionable alert rate for ASA alert types that deal with style issues generated by the ASA tool QA C.

AAIT method: The actionable alert rate for an alert type is the number of actionable alerts for the alert type divided by all alerts generated for the alert type. They evaluate two prioritization techniques: temporal coincidence and spatial coincidence. Temporal coincidence associates alerts with code changes. However, just because an alert is removed due to a code change does not mean that the alert was associated with the underlying anomaly fixed by the code change. Spatial coincidence reduces the noise from temporal coincidence by assessing the type of change made that removed an alert. Alerts are considered actionable if they are associated with changes due to fixing faults rather than other source code changes. The requirement for generating spatial coincidence is that changes in the source code that are checked into the repository should be associated with bugs listed in the bug database.

A version history graph is created for each file and is annotated with code changes on each edge. Alerts closed due to fixing a fault increment the alert count. After all versions of a file are evaluated, the remaining open alerts contribute to the overall count of

[8] Apache httpd is open source server software that may be found at: http://httpd.apache.org/.

[9] Wine is an open source program for running Windows applications on Linux, Unix, and other similar operating systems. Wine may be found at: http://www.winehq.org/.

[10] Apache httpd is an open source http web server. Apache httpd may be found at: http://httpd.apache.org/.

[11] rdesktop is a remote desktop client that may be found at: http://www.rdesktop.org/.

[12] wzdftpd is a FTP server that may be found at: http://www.wzdftpd.net/trac.

generated alerts. The actionable alert rate values can be used to prioritize alert types in future versions of software.

Method limitations: Boogerd and Moonen provide the following limitations to their work. The fault density measure lags behind the actual fault density due to differences between when a fault is introduced and reported. Major changes in file compositions may cause the correlations between rules with positive and negative relationships to change. Conservative estimates are used when calculating spatial coincidence to prevent underestimation. Since only one example is used, the results may not generalize to other software projects. Finally, technical problems may occur when building a project between platforms and when running ASA if the ASA generates an alert that can never happen.

Evaluation methodology: QA C reports alerts where C code violates the MISRA-C style rules. The experiment considers 214 revisions of an embedded mobile TV software (TVoM) project developed between August 2006 until June 2007. A bug database tied together the fault-fixes and alert closures used for calculating the actionable alert rate. Information about the bug reports were mined from the bug database, and had to meet the following requirements: (1) reports were a bug and not a functional change request; (2) associated with the C portions of the project; and (3) reports that were closed on or before June 2007.

Evaluation subjects: BM08B was evaluated using TV on Mobile (TVoM), described in Table 11.

Evaluation metrics: For each alert type, the actionable alert rate was calculated.

Evaluation results: The violation density and fault densities were plotted against each other with a negative trend line showing that fewer MISRA-C violations suggested a higher rate of faults. True positive rates were reported for 72 of the 141 possible MISRA-C rules. Boogerd and Moonen [6] found that "the true positive rates indicate that 15 out of 72 rules outperform a random predictor with respect to selecting fault-related lines."

6.6. CHECK 'N' CRASH and DSD-CRASHER

- Study: Csallner et al. [14]
- Artifact characteristics: DA
- ASA: ESC/JAVA [17]
- Dynamic Analysis: DAIKON [16] and JCRASHER [12]
- Language: Java
- AAIT type: Dynamic Detection (5.1.4)
- Research methodology: Other model comparison (5.2.2)

Objective: Csallner et al. [14] seek to "measure DSD-CRASHER against is a fully automated tool for moder object-oriented languages that finds bugs but produces no false bug warnings."

AAIT method: DSD-CRASHER incorporates a three step dynamic–static–dynamic process for identifying actionable static analysis alerts. The first dynamic step is utilizes DAIKON [16] which generates program invariants via execution of tests. These invariants are then used to refine the ASA step. ASA is conducted via the ESC/JAVA 2 [17] tool, which identifies locations for potential Java runtime exceptions. The alerts generated by ESC/JAVA 2 are input to JCRASHER [12], which automatically generates test cases with the goal of crashing the program and providing a concrete failure path to the runtime exception identified by ESC/JAVA 2.

Method limitations: For effective generation of invariants for the first dynamic step, DAIKON requires a set of regression tests that provide full coverage. The addition of DAIKON also reduces the scalability of the project. A final limitation is that DSD-CRASHER can only identify runtime exceptions and not other faults that may cause an undesired result.

Evaluation methodology: The evaluation of DSD-CRASHER explores the following questions: "(1) Can DSD-CRASHER eliminate some false bug warnings CHECK 'N' CRASH produces? And (2) Does DSD-CRASHER find deeper bugs than similar approaches that use a light weight bug search?" [14].

Evaluation subjects: JBoss JMS[13] and Groovy[14] are the subject programs for comparing CHECK 'N' CRASH with DSD-CRASHER, and details are provided in Table 11.

Evaluation metrics: DSD-CRASHER is evaluated by comparing the runtime and number of generated reports.

Evaluation results: When using DSD-CRASHER one fewer unactionable alert was reported on JBoss JMS in comparison with CHECK 'N' CRASH. DSD-CRASHER reduced the number of reported unactionable alerts by seven when analyzing Groovy in comparison with CHECK 'N' CRASH. Additionally, a comparison of DSD-CRASHER with ECLAT [42] found three class cast exceptions that ECLAT did not find for JBoss JMS and two additional alerts that ECLAT missed when analyzing Groovy. A final experiment investigated how well the ASA underlying CHECK 'N' CRASH and DSD-CRASHER, ESC/JAVA 2 [17], finds bugs seeded in open-source projects. The experiment considered three versions of Apache XML Security[15] containing 13–20 seeded bugs. Approximately half of the seeded bugs are unable to be found by ESC/JAVA 2, and the remainder that could be associated with alerts generated by ESC/JAVA 2 had no associated failing test cases generated by DSD-CRASHER.

6.7. HW09

- Study: Heckman and Williams [22]
- Artifact characteristics: AC, CC, SCR
- ASA: FINDBUGS [25]
- Language: Java
- AAIT type: Machine learning (5.1.6)
- Research methodology: Benchmark (5.2.3)

Objective: Heckman and Williams [22,23] present a process for using machine learning techniques to identify key artifact characteristics and the best models for classifying static analysis alerts for specific projects.

AAIT method: The process consists of four steps: (1) gathering artifact characteristics about alerts generated from static analysis; (2) selecting important, unrelated sets of characteristics; (3) using machine learning algorithms and the selected sets of characteristics to build models; and (4) selecting the best models using evaluation metrics.

Method limitations: The limitations to this process concern choosing the appropriate artifact characteristics and models for the classification of static analysis alerts.

Evaluation methodology: The machine learning-based model building process is evaluated on two FAULTBENCH [21] subject programs. The Weka [56] machine learning tool is used to generate artifact characteristic sets and classification models. The models are generated and evaluated by using 10 tenfold cross validations. Candidate models are created from 15 different machine learning techniques from five high-level categories: rules, trees, linear models, nearest neighbor models, and Bayesian models.

The hypothesis that static analysis alert classification models are program-specific is evaluated by comparing the important artifact characteristics and machine learning models.

[13] JBoss JMS is the messaging service component of the JBoss J2EE application server. JBoss may be found at: http://www.jboss.org.

[14] Groovy is a dynamic programming language for the Java Virtual Machine. Groovy may be found at: http://groovy.codehaus.org/.

[15] Information about Apache's XML Security module may be found at: http://santuario.apache.org/.

Evaluation subjects: The models are generated for two of the FAULTBENCH subject programs: jdom[16] and runtime[17].

Evaluation metrics: Heckman and Williams reported the accuracy, precision, and recall of the generated models.

Evaluation results: Only 50% of the selected artifact characteristics were common between the two projects. Additionally, models generated for jdom had 87.8% average accuracy while models generated for runtime had 96.8% average accuracy. The average recall and precision for jdom was 83.0% and 89.0%, respectively and 99.0% and 98.0% for runtime. The two best machine learning models were a version of nearest neighbor for jdom and decision tree for runtime, and, together with the attribute selection results, demonstrate that alert classification models are likely program-specific.

6.8. META-HEURISTIC

- Study: Rungta and Mercer [44]
- Artifact characteristics: CC
- ASA: JLINT [2]
- Language: Java
- AAIT type: Model checking (5.1.8)
- Evaluation Method: Random and optimal comparison (5.2.4)

Objectives: Rungta and Mercer [44] combine ASA and directed model checking to identify concurrency problems in Java programs. ASA, specifically JLINT, identifies potential areas of code where a concurrency error might exist. A META-HEURISTIC prioritizes the states in a model by analyzing the possible input sequences to the given state. The heuristic is used to drive identification of actual concurrency errors.

AAIT method: ASA provides a list of potential concurrency errors in a given program. A list of manually generated input sequences located near the ASA alert (i.e. a series of read and write operations that may cause a race condition) inform the META-HEURISTIC. The META-HEURISTIC is a value assigned to every state in the program and is the "number of locations in the input sequence that have been encountered along the current execution path." A secondary heuristic moves the analysis along by breaking ties between states that have the same META-HEURISTIC value. Three secondary heuristics are considered: (1) a polymorphic distance heuristic (PFSM) which calculates the distance between program locations; (2) a random heuristic; and (3) prefer-thread heuristic, which provides capability for a user to identify threads that they prefer to be analyzed. The greedy depth-first search algorithm mimics testing multi-threaded programs by checking the best child nodes (as measured by the META-HEURISTIC) for errors, until all nodes in the model have been checked. If an error is identified, the analysis has found an actual concurrency error and the alert generated by ASA is actionable.

Method limitations: No limitations are expressly listed, but the manual selection of input sequences to the META-HEURISTIC process is a threat to internal validity of the process.

Evaluation methodology: Rungta and Mercer evaluate their META-HEURISTIC technique on six multi-threaded Java programs. They compare their META-HEURISTIC to a random depth-first search. Additionally, three different secondary heuristics are considered. In the evaluation, 100 trials of each search are run to mitigate the randomness associated with backtracking during the depth-first searches. For the analysis, the alerts were generated using JLINT and Java Pathfinder [51] served as the model checker.

Evaluation subjects: Six evaluation subjects were used: three from a benchmark created by IBM Research Lab in Haifa and three from classical concurrency problems. The subjects are described in Table 12.

Evaluation metrics: Rungta and Mercer used the error density metric to evaluate their META-HEURISTIC against a random depth-first search. Additionally, the different secondary heuristics were evaluated against each other by the number of states in the generated model.

Evaluation results: The META-HEURISTIC technique performed better than the random depth-first search for all trials. Additionally, the PFSM secondary heuristic performed the same as or better than the other secondary heuristics for all trials. The PFSM secondary heuristic reported an error density of 1.0 for all trials. In most trials, PFSM performed better in the number of states generated when compared to the other secondary heuristics.

6.9. INTFINDER

- Study: Chen et al. [8]
- Artifact characteristics: DA
- ASA: UNSPECIFIED
- Language: C
- AAIT type: Dynamic detection (5.1.7)
- Evaluation Method: Other baseline comparison (5.2.1)

Objectives: Chen et al. [8] propose an approach to find integer faults in binary programs via a combination of static and dynamic analyses to achieve the following: (1) find additional integer faults beyond integer overflow, (2) recreate precise types during decompilation, (3) lower the number of ASA generated false positives, and (4) reduce the time to run dynamic analysis while also minimizing false negatives.

AAIT method: INTFINDER decompiles binary executables into a form of static single assignment (SSA). The decompiler's type analysis is not sufficient; therefore, an additional type analysis is done based on trends observed when mining the Common Vulnerability and Exposures (CVE) database.[18] The type analyses identify integer alerts associated with memory allocation, array indices, memory copies, and signed upper bound checks. With the additional type information, alerts are generated for a set of instructions where there are potential integer faults. Taint analysis and a dynamic detection tool identify actual integer faults from the alerts.

Method limitations: The limitations of INTFINDER are associated with decompilation, dynamic detection, and test input creation. Not all functions are decompiled, which may lead to incomplete analyses. Additionally, decompilation has any limitations of the underlying tool (in this case Boomerang[19]). Dynamic detection limitations are associated with how semantics are understood. Most logic operations were not considered due to difficulty in identifying integer faults. The final limitation is that the test inputs are from known vulnerabilities which can only identify known vulnerabilities.

Evaluation methodology: INTFINDER was evaluated on six programs with known integer vulnerabilities reported in the CVE. First, the preciseness of the ASA portion of INTFINDER was evaluated by comparing the number of instructions with a precise type to the number of suspect instructions. infinder was also evaluated by how well the tool identified integer faults and by how well the tool performed.

Evaluation subjects: Six evaluation subjects were used. The subjects have known vulnerabilities as reported in the CVE.

[16] jdom is an XML library, and may be found at: http://www.jdom.org.

[17] Runtime is the org.eclipse.runtime package from the Eclipse project. Information on Eclipse may be found at: http://eclipse.org/.

[18] The Common Vulnerabilities and Exposures database lists known vulnerabilities. The CVE may be found at: http://cve.mitre.org/.

[19] Boomerang decompiles programs for the x86 platform. Boomerang may be found at: http://boomerang.sourceforge.net/.

Evaluation metrics: INTFINDER was evaluated using the following metrics: preciseness of type analysis, number of false positives, and time to execute.

Evaluation results: An average preciseness of 97.1% is reported demonstrating a reduction in false positives. INTFINDER was able to find all 16 of the reported vulnerabilities with no false positives. Additionally, INTFINDER found a new bug in one of the programs. Finally, performance was measured comparing the use of Pin alone with INTFINDER. Pin had a 3.7× average overhead and INTFINDER had an 4.4× average overhead.

6.10. FIS

- Study: Yu et al. [59]
- Artifact characteristics: CC
- ASA: UNSPECIFIED
- Language: Java
- AAIT type: Contextual information (5.1.2)
- Evaluation Method: Other (5.2.6)

Objective: Yu et al. [59] propose using a fuzzy inference system (FIS) to identify the areas of a program under analysis that are more suited for ASA or model checking; therefore reducing ASA false positives and model checking's state explosion problem.

AAIT method: The FIS process consists of four steps: (1) classification of security faults, extracting keyword information about the faults, and clustering the program into smaller units for faster analysis; (2) fuzzy inference that maps the keywords to Java constructs and applies the mapping to determine which of type of analysis to use; (3) running ASA on the portions of the code identified to work best with ASA; and (4) running model checking on the portions of the code identified to work best with model checking.

Method limitations: The ASA used in FIS had a high performance overhead, which is a limitation of the technique. Yu et al. proposed four techniques to reduce the performance overhead including parallel computing, sorting of the generated abstract syntax trees to take advantage of similarities, extracting abstract syntax trees so that one rule can be loaded at a time, and load similar rules together.

Evaluation methodology: FIS is implemented as an Eclipse plugin. Evaluation of FIS consisted of measuring how well the tool detected injected faults and how long the tool ran. FIS was compared with model checking and ASA run alone.

Evaluation subjects: Yu et al. evaluated FIS on a peer-to-peer client with injected vulnerabilities. Additionally, four other open-source projects were used as subject programs as described in Table 11.

Evaluation metrics: FIS was evaluated using the following metrics: number of faults found, execution time, and time distributions.

Evaluation results: In the peer-to-peer client with seeded faults, FIS identified 73 or the 74 seeded vulnerabilities. Model checking alone identified 27 and ASA alone identified 53. The hybrid approach of FIS performed better than the individual parts alone. The performance of FIS was slower than ASA, but faster than model checking, and the performance enhancements to ASA generated on average a 50% improvement. A majority of the time to run FIS was associated with model checking followed by the clustering algorithm. A comparison of the faults found by the ASA and model checking portions of FIS show that the two techniques are complementary.

6.11. Classification results discussion

Ogasawara et al. [40] first demonstrate that ASA is useful for identifying problems in source code and that minimizing the alerts reported by ASA through some filtering mechanism can increase the effectiveness of using ASA. Ogasawara et al.'s work explains the most basic AAIT: selection of alert types of interest from experience with the code base. Selection of 41 alert types of interest leads to 83% reduction in reported alerts. CHECK 'N' CRASH, DSD-CRASHER [14], and SCAS [57] were successful in reducing reported alerts on evaluated subject programs through more programmatic reduction techniques. DSD-CRASHER saw a 41.1% reduction in reported alerts when compared to CHECK 'N' CRASH. However, the alert reduction provided by DSD-CRASHER missed one of the actionable alerts reported by CHECK 'N' CRASH, which shows that DSD-CRASHER is not a safe technique. SCAS averaged a 32.3% alert reduction across the three branches of the subject program, but there is no discussion of FNs due to the reduction. INTFINDER [8] used dynamic detection to correctly identify instructions in binary executables that contained integer faults with no false positives.

The AAIT proposed by Aggarwal et al. [1] and FIS [59] identifies places in the code where dynamic analysis or model checking should be used in place of static analysis, which could lead to a reduction of reported alerts. Aggarwal et al. [1] did not report any numerical results to support their hypothesis. Yu et al. [59] found that ASA and model checking were complementary techniques and found different types of faults. Rungta and Mercer [44] used model checking in combination with ASA, but unlike Yu et al., the alerts generated by ASA were used to guide model checking rather than having model checking as a complementary technique.

AAITs proposed by Boogerd and Moonen [6], Heckman and Williams [22], and Williams and Hollingsworth [55] were evaluated using precision, which is a measure of the number of actionable alerts correctly identified out of all alert predicted as actionable. The BM08B AAIT reported precision ranging from 5.0% to 13.0%, which is lower than the precisions reported when using HISTORYAWARE and HW09 AAITs. HISTORYAWARE AAIT had a higher precision than the NAÏVERANKING AAIT, but lower precision than HW09. Precision is a metric commonly reported by many of the classification and prioritization studies. The precision numbers for the above three studies are summarized with the prioritization precision data in Table 12.

The static analysis tools and subject program languages used varied by study, which only allows for some general comparison of the results. However, the overall results of classification studies support the use of AAIT to classify actionable alerts.

7. Prioritization results

Prioritization AAITs order alerts by the likelihood an alert is an indication of an actionable alert [21]. The subsections below describe 11 prioritization AAITs using a similar reporting template used for the classification metrics in Section 6.

7.1. Z-RANKING

- Study: Kremenek and Engler [36]
- Artifact characteristics: CC
- ASA: MC [15]
- Language: C
- AAIT type: Mathematical and statistical models (5.1.5)
- Research methodology: Random and optimal comparison (5.2.1)

Objective: Kremenek and Engler [36] proposed a statistical model for prioritizing static analysis alerts.

AAIT method: Unlike most other ASA, the MC [15] tool reports "(1) the locations in the program that satisfied a checked property [successful checks] and (2) locations that violated the checked property [failed checks]" [36]. The z-ranking statistical technique is built on the premise that alerts (represented by failed checks), identified by the same detector, and associated with many successful checks are likely actionable. Additionally, a "strong hypothesis" proposes that unactionable alerts are associated with many other unactionable alerts. The special case of the "strong hypothesis" is called the "no-success hypothesis" and states "[alerts] with no coupled successful checks are exceptionally unlikely [actionable alerts]." The "no-success hypothesis" will not hold if the "strong hypothesis" does not hold. A hypothesis test is run on the proportion of successful checks out of all reports for a given grouping of checks. Alerts are grouped by some artifact characteristic, called a *grouping operator*, they all share (e.g. call site, number of calls to free memory, function). The hypothesis testing allows for consideration of the size of each possible grouping of checks, and the final number is called a z-score. Alerts generated by static analysis are prioritized by their z-score.

Method limitations: A limitation of the z-ranking technique is that the prioritization's success depends on the grouping operator.

Evaluation methodology: The alert prioritization of the z-ranking technique is compared to an optimal and random ranking of the same alerts. A hypergeometric distribution is used to generate the random ordering of alerts. The ranking of the alerts are evaluated for three types of detectors in the mc ASA system: lock error detectors (identify concurrency faults), free error detectors (identify dereferences of freed memory), and string format error detectors (identify faults associated with string format libraries).

Evaluation subjects: z-ranking is run on two subject programs: the Linux kernel and a commercial system, System X. Both are described in Table 11.

Evaluation metrics: The cumulative number of bugs discovered is plotted on a graph for each inspection.

Evaluation results: For the lock errors, 25.4–52.2% of the alerts required inspection before finding all actionable alerts. For the free errors and string format errors, the first 10% of the ranked alerts found 3.3 times and 2.8 times the bugs than the first 10% of randomly ordered alerts. A set of 1.0×10^5 randomly generated orderings of the alerts were compared to the z-ranking prioritization. At most, 1.5% of the random orderings were better than alerts ordered by z-ranking.

7.2. feedback-rank

- Study: Kremenek et al. [35]
- Artifact characteristics: AC
- ASA: mc [15]
- Language: C
- AAIT type: Machine learning (5.1.6)
- Research methodology: Random and optimal comparison (5.2.1)

Objective: Based on the intuition that alerts sharing an artifact characteristic tend to be either all actionable or all unactionable, Kremenek et al. [35] developed an adaptive prioritization algorithm, feedback-rank.

AAIT method: Each inspection of an alert by a developer adjusts the ranking of uninspected alerts. After each inspection, the set of inspected alerts are used to build a Bayesian Network, which models the probabilities that groups of alerts sharing a characteristic are actionable or unactionable. Additionally, a value representing how much additional information inspecting the report will provide to the model is generated for each alert. The information gain value is used to break ties between alerts with the same probability of being an anomaly.

Method limitations: No limitations are explicitly stated.

Evaluation methodology: Alerts ordered by feedback-rank are compared to the optimal and random ordering of the same alerts. For the feedback-rank algorithm, they consider two alert prioritization schemes. In one prioritization scheme, there is no information about already inspected alerts to build the model. The model is updated as alerts are inspected, which represents a project just starting to use ASA. The other prioritization scheme considers a set of alerts as already inspected, and uses the classifications from those alerts to build the initial model, which could potentially lead to a better initial prioritization of alerts. Three subsets of a subject's alerts generate three models, in particular the conditional probability distribution of the Bayesian Network: the entire code base, self-trained, and a 90% reserved model. For the Bayesian Network trained on the entire code base, all of the generated alerts and their classifications are used to build the conditional probability distribution of the actionable and unactionable alerts. For the self-trained set, the conditional probability distribution values are trained on the set of alerts that are also ranked by the Bayesian Network. Finally, in the 90% reserved model, 90% of the alerts are used to train the conditional probability distributions for the Bayesian Network and the model is tested on the remaining 10% of alerts.

Evaluation subjects: feedback-rank is evaluate on Linux and a commercial system called System X, both of which are described in Table 11.

Evaluation metrics: A custom metric, performance ratio, allows for comparison between the rankings generated via feedback-rank technique and the random ordering of alerts. Performance ratio is the ratio between random and the ranking techniques' "average inspection 'delay' or 'shift' per bug from optimal."

Evaluation results: The results show that all detectors in the mc system show a 2–8× improvement of performance ratio over random when using feedback-rank. The self-trained model for the ASA tool, Alock (part of the mc system), showed a 6–8× improvement of performance ratio over random when seeded with partial knowledge of some alert classifications.

7.3. jks05

- Study: Jung et al. [30]
- Artifact characteristics: CC
- ASA: airic [30]
- Language: C
- AAIT type: Machine learning (5.1.6)
- Research methodology: Train and test (5.2.5)

Objective: Jung et al. [30] use a Bayesian network to generate the probability of an actionable alert.

AAIT method: The probability of an actionable alert is generated given a set of 22 code characteristics (e.g. syntactic and semantic code information like nested loops, joins, and array information). The model is generated via inspected alerts generated on the Linux kernel code and several textbook C programs. A user-specified threshold limits the number of alerts reported to developers, which reduces the set of alerts for a developer to inspect.

Method limitations: The artifact characteristics that serve as input to the Bayesian network influence how well the model will predict actionable alerts.

Evaluation methodology: The train and test technique was used to evaluate the proposed prioritization technique. The alerts were randomly divided into two equal sets. One set was used to train a Bayesian model using the artifact characteristics (called symptoms) generated for each alert, and the model was tested on the second set of alerts. The selection of training and test sets and model building was repeated 15 times.

Evaluation subjects: Evaluation subjects were "... some parts of the Linux kernel and programs that demonstrate classical algorithms." Therefore, the evaluation subjects are not specifically listed in Table 11.

Evaluation metrics: Jung et al. report the accuracy, precision, and recall of their analysis.

Evaluation results: The precision, recall, and accuracy are 38.7%, 68.6%, and 73.7%, respectively. Additionally, 15.17% of the unactionable alerts were inspected before 50% of the actionable alerts were inspected. Jung et al. observe that if the threshold for "trueness" is lowered, then all actionable alerts will be provided to the user at the cost of an unknown additional amount of unactionable alerts.

7.4. ALERTLIFETIME

- Study: Kim and Ernst, 2007a [31]
- Artifact characteristics: AC and SRC
- ASA: FINDBUGS [25], PMD,[20] JLINT [2]
- Language: Java
- AAIT type: Mathematical and statistical models (5.1.5)
- Research methodology: Other baseline comparison (5.2.4)

Objective: Kim and Ernst [31] prioritize alert types by the average lifetime of alerts sharing the type.

AAIT method: The premise is that alerts fixed quickly are more important to developers. The lifetime of an alert is measured at the file level from the time the first instance of an alert type appeared until closure of the last instance of that alert type. Alerts that remain in the file at the last studied revision are given a penalty of 365 days added to their lifetime.

Method limitations: Lack of alert tracing when line and name changes occur leads to error in the alert lifetime measurement and the variance of lifetimes for an alert type is unknown. The technique assumes that important problems are fixed quickly; however, alerts that are fixed quickly may be the easiest bugs to fix and not the most important alerts [31].

Evaluation methodology: Validation of the ALERTLIFETIME AAIT compared the alerts ordered by lifetime with alerts ordered by the tool specified severity.

Evaluation subjects: Kim and Ernst [31] evaluated their technique on Columba[21] and jEdit,[22] which are listed in Table 11.

Evaluation metrics: Kim and Ernst [31] reported the prioritization and severity of the top 10 and bottom 10 alerts for each project. Additionally, they reported the correlation between the alert lifetimes for the two subject programs.

Evaluation results: Results showed that the alert lifetime prioritization did not correspond to the tool specified severity. Comparison of the alert type lifetimes between the two subject programs had a correlation coefficient of 0.218, which demonstrates that the alert type ordering for one program may not be applicable for another program.

7.5. HWP

- Study: Kim and Ernst, 2007b [32]
- Artifact characteristics: AC and SCR
- ASA: FINDBUGS [25], PMD, JLINT [2]
- Language: Java
- AAIT type: Mathematical and statistical model (5.1.5)
- Research methodology: Other baseline comparison (5.2.4)

Objective: Kim and Ernst [32] use the commit messages and code changes in the source code repository to prioritize alert types.

AAIT method: The history-based warning prioritization (HWP) weights alert types by the number of alerts closed by fault- and non-fault-fixes. A fault-fix is a source code change where a fault or problem is fixed (as identified by a commit message) while a non-fault-fix is a source code change where a fault or problem is not fixed, like a feature addition. The initial weight for an alert type is zero. At each fault-fix the weight increases by an amount, α. For each non-fault-fix, the weight increases by $1 - \alpha$. The final step normalizes each alert type's weight by the number of alerts sharing the type. A higher weight implies that alerts with a given type are more likely to be actionable.

Method limitations: The prioritization technique considers all alert sharing the same type in aggregate, which assumes that all alerts sharing the same type are homogeneous in their classification. The prioritization fails for alerts generated in later runs of ASA if the alert type never appears in earlier versions of the code.

Evaluation methodology: Evaluation of the proposed fix-change prioritization trained the model using the first $(n/2) - 1$ revisions and then tested the model on the latter half of the revisions.

Evaluation subjects: HWP was evaluated on three subject programs, Columba, Lucene,[23] and Scarab,[24] which are described in Table 11.

Evaluation metrics: The precision of the tool's alert prioritization with the prioritization of alerts based on the project's history were compared.

Evaluation results: The best precision for the three subject programs (in the order listed above) is 17%, 25%, and 67%, respectively when using HWP as compared to 3%, 12%, and 8%, respectively when prioritizing the alerts by the tool's severity or priority measure. Additionally, when only considering the top 30 alerts, the precision of the fix-based prioritization is almost doubled, and in some cases tripled, from the tool's ordering of alerts.

7.6. ISA

- Study: Kong et al. [34]
- Artifact Characteristic: AC
- ASA: RATS,[25] ITS4 [50], FLAWFINDER[26]
- Language: C
- AAIT type: Data fusion (5.1.7)
- Research methodology: Other model comparison (5.2.2)

Objective: Kong et al. [34] use data fusion to identify vulnerable code using alerts generated by ASAs focused on finding security vulnerabilities.

AAIT method: The ISA tool reports a score for each aggregated alert type, which represents the likelihood the alert is a vulnerability. The score is the combination of the tool's alert severity and the contribution of each tool summed across all tools. The feedback from the user when inspecting alerts contribute to the weights associated with a specific ASA.

Method limitations: The technique is limited by the mapping of alerts between tools.

Evaluation methodology: The prioritization of the ISA AAIT is compared with the prioritization of the individual ASA tools that

[20] PMD is ASA for Java: http://pmd.sourceforge.net/.

[21] Columba is a open-source email client. Columba may be found at: http://sourceforge.net/projects/columba/.

[22] jEdit is an Java based text editor. jEdit may be found at: http://www.jedit.org/.

[23] Lucene is a open-source search engine. Lucene may be found at: http://lucene.apache.org/.

[24] Scarab is an open-source issue tracker. Scarab may be found at: http://scarab.tigris.org/.

[25] RATS is ASA for C, C++, Perl, and Python developed by Fortify Software: http://www.fortify.com/security-resources/rats.jsp.

[26] FLAWFINDER is ASA for C: http://www.dwheeler.com/flawfinder/.

make up the ISA tool on subject programs with known vulnerabilities.

Evaluation subjects: All of the subject programs used by Kong et al. in Table 11, have known vulnerabilities, which provide a measure of how well ISA and the individual ASA tools perform.

Evaluation metrics: ISA is evaluated by the number of false positives and false negatives. Additionally, the efficiency of the tool is compared with the individual ASA tools.

Evaluation results: The results show that ISA has a lower rate of false positives and false negatives than the individual ASA for two of the three subject programs. Additionally, ISA is found to be more efficient (defined as the likelihood of finding a vulnerability when inspecting the alerts) than the individual static analysis tools.

7.7. YCK07

- Study: Yi et al. [58]
- Artifact characteristics: CC
- ASA: AIRIC [30]
- Language: C
- AAIT type: Machine learning (5.1.6)
- Research methodology: Train and test (5.2.5)

Objective: Yi et al. [58] compare the classification of actionable and unactionable alerts for several machine learning algorithms.

AAIT method: The static analysis tool AIRAC, finds potential buffer overrun vulnerabilities in C code. There are three types of symptoms: syntactic, semantic, and information about the buffer uncovered by static analysis. The process for building the linear regression model considered attribute subset selection to minimize collinear attributes.

Eight machine learning techniques prioritize static analysis alerts into actionable and unactionable groups. The symptoms about the alerts are the independent variables and the classification of the alert is the dependent variable. The alerts are divided into a training and test set using an approximately two-thirds one-third split. The training-test cycle is repeated 100 times. The results are summed over all 100 models. The open-source statistical program R[27] was used to train and test the models.

Method limitations: Some of the symptoms as input to the model are correlated, which may cause some of the models to be ineffective.

Evaluation methodology: Overall, there were 332 alerts generated for all of the subject programs. The different machine learning techniques were evaluated by comparing the AUC of ROC curves. The closer the area is to one, the better the performance of the model.

Evaluation metrics: The YCK07 models used the AUC of ROC curves for evaluation. Additionally, the number of unactionable alerts inspected before the first percentage of actionable alerts were identified is reported.

Evaluation subjects: The YCK07 models were evaluated on 36 files and 22 programs, the details of which are not provided.

Evaluation results: The AUC for the ROC curves varied from 0.87 to 0.93. Additionally, only 0.32% of the unactionable alerts were identified before the first 50% of the actionable alerts. Also, 22.58% of the actionable alerts were inspected before the first unactionable alert was inspected.

7.8. ELAN, EFAN$_H$, and EFAN$_V$

- Study: Boogerd and Moonen, 2008a
- Artifact characteristics: CC
- ASA: UNSPECIFIED
- Language: C
- AAIT type: Graph theory (5.1.3)
- Research methodology: Other baseline comparison (5.2.4)

Objective: Boogerd and Moonen [7] prioritize alerts by execution likelihood [4] and by execution frequency [7].

AAIT method: Execution likelihood is defined as "the probability that a given program point will be executed at least once in an arbitrary program run" [7]. Execution frequency is defined as "the average frequency of [program point] v over all possible distinct runs of [program] p" [7]. Alerts with the same execution likelihood are prioritized the same, but may actually have varying importance in the program. Execution frequency solves the limitation of execution likelihood by providing a value of how often the code will be executed [7].

Prediction of the branches taken when calculating the execution likelihood and frequency are important to the Execution Likelihood ANalysis (ELAN) and Execution Frequency ANalysis (EFAN) techniques [7]. The ELAN AAIT (introduced in [4]) traverses the system dependence graph of the program under analysis and generates the execution likelihood of an alert's location and heuristics are used for branch prediction. There are two variations of the EFAN AAIT: one uses heuristics for branch prediction based on literature in branch prediction (EFAN$_H$) and the other uses value range propagation (EFAN$_V$). Value range propagation estimates the values of variables from information in the source code.

Method limitations: The graphs generated may be missing paths or have infeasible paths which represent false negatives and false positives, respectively.

Evaluation methodology: The effectiveness of the ELAN and EFAN prioritization techniques were compared with execution data, gathered by automated regression test runs, and not with the actual actionability of ASA alerts.

Evaluation subjects: Five open-source programs as listed in Table 11, were used in the case study to compare ELAN and EFAN.

Evaluation metrics: Wall's unweighted matching method [54] compares the prioritized list of alerts with the list of alerts ordered by the actual execution data and produces a measure of correlation.

Evaluation results: The ELAN AAIT had an average correlation of 0.39 with the actual execution values for the top 10% of alerts, which outperformed the EFAN$_H$ and EFAN$_V$ with correlations of 0.28 and 0.17, respectively. One limitation of the work is that the created system dependence graph may miss dependencies, which could lead to missing potential problems. Additionally, dependencies that are actually impossible to traverse introduce unactionable alerts.

7.9. APM

- Study: Heckman and Williams, 2008
- Artifact characteristics: AC
- ASA: FINDBUGS [25]
- Language: Java
- AAIT type: Mathematical and statistical models (5.1.5)
- Research methodology: Benchmarks (5.2.3)

Objective: Heckman and Williams [21] adaptively prioritize individual alerts using the alert's type and location in the source code.

AAIT method: The adaptive prioritization model (APM) re-ranks alerts after each developer inspection, which incorporates feedback about the alerts into the model. The APM ranks alerts on a scale from −1 to 1, where alerts close to −1 are more likely to be

[27] R is an open-source statistical program: http://www.r-project.org/.

unactionable and alerts close to 1 are more likely to be actionable. Alerts are prioritized by considering the developer's feedback, via alert fixes and suppressions, to generate a homogeneity measure of the set of alert's sharing either the same alert type or code location. APM relies on developer feedback to prioritize alerts. Three prioritization models were considered that focused on different combinations of artifact characteristics: (1) alert type (ATA), (2) alert's code location (CL), and (3) both alert type and the alert's code location (ATA + CL).

Method limitation: An assumption of the model is that alerts sharing an alert type or code location are likely to be all actionable or all unactionable. If the assumption does not hold, then the prioritization of actionable alerts will likely be poor.

Evaluation methodology: The three prioritization models were compared with the tool's ordering of alerts, an optimal ordering of alerts, and with each other.

Evaluation subjects: Three versions of APM were evaluated on the six subject programs of the FAULTBENCH benchmark as listed in Table 11.

Evaluation metrics: APM was evaluated with accuracy, precision, recall, and the correlation with an optimal ordering. Additionally, evaluation of the APM models used a variation of a ROC curve, called the anomaly detection rate curve. The anomaly detection rate curve measures the percentage of anomalies detected against the number of alert inspections and is similar to the weighted average of the percentage of faults detected measure used by Rothermel et al. [43].

Evaluation results: The anomaly detection rate curve for the APM prioritization models was larger (53.9–72.6%) than the alerts ordered by when FINDBUGS identified them (TOOL – 50.4%). Additionally, comparing the prioritization generated by each of the techniques with an optimal prioritization demonstrated a moderate to moderately strong correlation (0.4–0.8) at a statistically significant level for five of the six subject programs. The average accuracy for the three APM models ranged from 67% to 76% and the model that prioritized alerts by the alert's type (ATA) was found to perform best overall.

7.10. MMW08

- Study: Meng et al. [38]
- Artifact characteristics: AC
- ASA: FINDBUGS [25], PMD, JLINT [2]
- Language: Java
- AAIT type: Data fusion (5.1.7)
- Research methodology: Other (5.2.6)

Objective: Meng et al. [38] propose an approach that merges alerts that are common across multiple static analysis tools run on the same source code.

AAIT method: The combined alerts are first prioritized by the severity of the alert and are then prioritized by the number of tools that identify the alerts. A map associates alerts for a specific tool to a general alert type.

Method limitations: The MMW08 technique has the same limitations as Kong et al. [34].

Evaluation methodology: They run FINDBUGS [25], PMD, and JLINT [2] on the subject program.

Evaluation subjects: Evaluation of the MMW08 technique was on a small, UNNAMED, subject program, which is not listed in Table 11.

Evaluation metrics: MMW08 was evaluated by the number of alerts generated.

Evaluation results: Meng et al. [38] report four of the alerts generated for the small subject program, one of which was reported by two tools.

7.11. RPM08

- Study: Ruthruff et al. [45]
- Artifact characteristics: AC, CC, SCR
- ASA: FINDBUGS [25]
- Language: Java
- AAIT type: Mathematical and statistical models (5.1.5)
- Research methodology: Other model comparison (5.2.2)

Objective: Ruthruff et al. [45] use a logistic regression model to predict actionable alerts.

AAIT method: Thirty-three artifact characteristics are considered for the logistic regression model. Reducing the number of characteristics for inclusion in the logistic regression model is done via a screening process, whereby logistic regression models are built with increasingly larger portions of the alert set. Characteristics with a contribution lower than a specified threshold are thrown out until some minimum number of characteristics remains. Two models were considered: one for predicting unactionable alerts and the other for predicting actionable alerts. For the actionable alerts model, two specific models were built: one considered only alerts identified as actionable and the second considered all alerts.

Method limitations: One limitation is that the screening window may be the incorrect size for the code base. Additionally, some of the artifact characteristic values were measured at the time of the screening, not at the time when the alert was generated, which may not reflect actual conditions for alert generation. Finally, some of the artifact characteristics are collinear, but collinear factors are not excluded from the generated model.

Evaluation methodology: Evaluation of the RPM08 models compared the generated models with a modified model from related work in fault identification and a model built using all of the suggested alert characteristics. The related work models use complexity metrics to predict faults and come from the work by Bell, Ostrand, and Weyuker [3,41]. Ruthruff et al. [45] adapt the models, which they call BOW and BOW+, for alert prioritization. The BOW model is directly from the work by Bell et al. The BOW+ models additionally included two static analysis metrics, the alert type and the alert priority, in addition to the complexity metrics.

Evaluation subjects: RPM08 was evaluated on 1652 alerts from Google's bug database, as shown in Table 11.

Evaluation metrics: The cost in terms of time to generate the data and build the models was compared in addition to the precision of the predictions.

Evaluation results: The APM08 model building technique, which used screening, took slightly less than seven hours to build and run, which is reasonable compared to the 5 days required to build the APM08 model using all available data. However, the proposed model takes a longer time than the BOW and BOW+ models. The precision of the screening models ranged from 73.2% to 86.6%, which was higher than the BOW and BOW+ models with precision between 60.9% and 83.4%, especially when predicting actionable warnings.

7.12. Prioritization results discussion

The ELAN, EFAN_H, and EFAN_V AAIT developed by Boogerd and Moonen [4,7] found that the execution likelihood of alert locations was highly correlated with actual execution, which is encouraging that their prioritization could be used as a measure of alert severity. However, the results do not address the accuracy of their prioritization in identifying actionable alerts, and thus cannot be compared to the other prioritization models.

Kremenek and Engler [36] demonstrate the efficacy of Z-RANKING in identifying actionable alerts for three types of ASA alert detectors compared to a random ordering of alerts. The lock ASA finds three types of locking inconsistencies. Z-RANKING identified three, 6.6, and four times more actionable alerts than the random baseline in the first 10% of inspections for the three types of locks in Linux. Z-RANKING showed a 6.9 times improvement over the random baseline for detecting lock faults for System X. FEEDBACK-RANK [35] showed up to an eight time improvement in actionable alert identification over a random baseline. The evaluation of FEEDBACK-RANK used eight ASA tools that each identifies one type of alert. The average improvement of FEEDBACK-RANK was 2–3 times better than random. The use of Z-RANKING or FEEDBACK-RANK improves the prioritization of alerts when compared to the random ordering of alerts. The results from Kremenek et al. [35,36] provide empirical evidence for the efficacy of prioritization techniques; however, other prioritization techniques did not report the same metrics, precluding comparison.

Jung et al. [30], Kim and Ernst [32], Yi et al. [58], Heckman and Williams [21], and Ruthruff et al. [45] report one or more classification (5.4) and/or prioritization metrics (5.5), but specifically all of the studies, except Yi et al., report precision. Classification metrics may be used on prioritization AAITs when there is a cutoff value that separates alerts likely to be actionable from those likely to be unactionable. Table 12 summarizes the five studies that report precision. Ruthruff et al. [45] demonstrate the highest precision with their RPM08 AAIT. Heckman and Williams' [21] APM AAIT had 67.1–76.6% accuracy with comparatively low precisions. The low precision results from averaging the precision generated at every inspection. Precision is low when there are few TP predictions, which can occur when the model is incorrect, or when all TP alerts have been inspected. Kim and Ernst [31] prioritize alert types by their lifetime and compare the prioritization with the tools' prioritization. Additionally, the alert type prioritizations are compared between two subject programs and have a low correlation of 0.218. Yi et al. [58] compare prioritization techniques using the AUC for Response Operating Characteristic curves. The boosting method had the largest area under the curve at 0.9290.

Kong et al. [34] show that the alert prioritization generated via data fusion of redundant alerts performed better than the alert prioritization's of the individual tools. Meng et al. [38] discussed some of the alerts found by the ASA; however, there were no numerical results on a larger subject program.

Similarly to the classification results, the static analysis tools and subject program languages used varied by study, which, again, only allows for some general comparison of the results. However, the overall results support the use of prioritization AAITs to prioritize actionable alerts.

8. Combined discussion

The results presented in the selected studies and summarized in Sections 6.8 and 7.12 support the premise that AAIT supplementing ASA can improve the prediction of actionable alerts for developer inspection, with a tendency for improvement over baseline ordering of alerts. Analyzing the combined results answer RQ3.

- RQ3: What conclusions can we draw about the efficacy of AAITs from results presented in the selected studies?

Seven of the 21 studies presented results in terms of precision: the proportion of correctly classified anomalies (TP) out of all alerts predicted as anomalies (TP + FP). When no AAIT is available, we can consider all unactionable alerts as FPs. Therefore, the precision of the ASA is the proportion of all actionable alerts out of the unactionable alerts. When using an AAIT, the precision then becomes the number of actual actionable alerts (those alerts the developer fixes) out of all alerts predicted to be actionable. Table 12 presents the combined precision data from seven of the classification and prioritization AAITs, where data are available.

The precisions reported in Table 12 varied from 3% to 98%, which demonstrate the wide variability of the AAIT used to supplement ASA. Direct comparisons of the precision are difficult to make due to the different subject programs used for evaluation and the different methods used for measuring precision within the study. For example, the precision of APM was the average precision of the prioritization after each alert inspection. The precision after many of the inspections was very low because all actionable alerts were predicted to be unactionable due to the overwhelming number of unactionable alerts in the alert set [21]. The other AAIT calculated precision from the initial classification or prioritization.

Figs. 2 and 3, from the data in Table 12, show the size of the selected studies' evaluated subject programs in terms of KLOC and number of alerts versus the precision of the subject and factor. A factor is the AAIT, which represents the process variable of the experimental evaluation [49]. The precision varies greatly across the size of the subject program in terms of both the KLOC and number of alerts for the program. The AAITs that produce the highest precision for a specific subject program are both from research done by Heckman and Williams [22] where a process to find the most predictive artifact characteristics and machine learning model were applied to the two subject programs jdom and runtime. The precision for these two subject programs was 89% and 98%, respectively. Ruthruff et al. [45] reported the next highest precision of 78% when used a statistical screening process to identify the most predictive artifact characteristics on a randomly selected sets of alerts generated on Google's java code.

BMO8B and APM showed the lowest precision during evaluation. Precision is low when the actionable alert predictions are incorrect

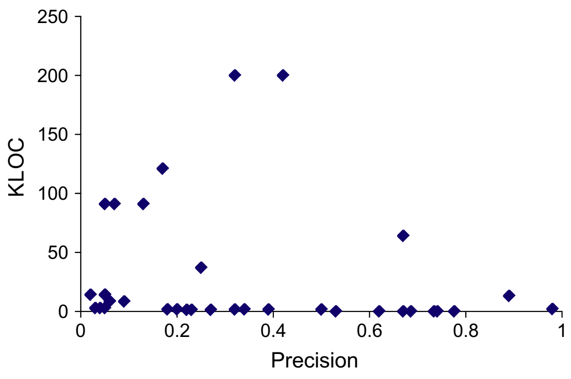


Fig. 2. The precision reported for each of the subject-factor pairs for the seven studies that reported precision against KLOC.

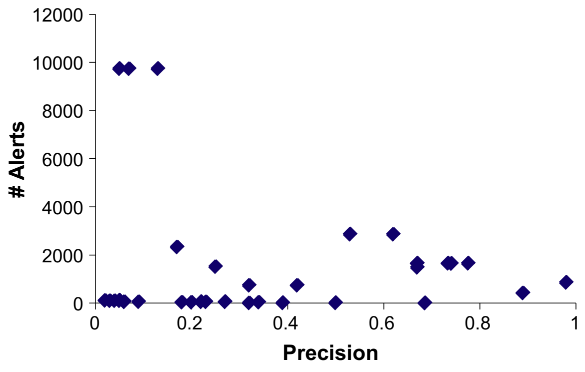


Fig. 3. The precision reported for each of the subject-factor pairs for the seven studies that reported precision against the number of alerts.

and there are many more FPs than TPs. Boogerd and Moonen [6] report low precision when evaluating BM08B because there are many more FP alerts reported for an alert type than there are actual violations in practice. Heckman and Williams [21] average the precision for each alert inspection in their prioritized list of alerts. If there were no correctly predicted actionable alerts for an inspection, the precision was zero for that inspection, which can drop the average precision across all inspections for a subject program. The reported accuracy of 67–79% when using APM is more encouraging to the usefulness of APM.

The high precision reported by some of the seven studies reported in Table 12 is encouraging that using AAITs to supplement ASA identifies actionable alerts. Of those alerts identified as potentially actionable by an AAIT, on average 34% of the alerts were actionable. However, these results to do not identify the actionable alerts that were missed by an AAIT, which is measured by recall. Additionally, these results do not identify how well an AAIT was able to differentiate between actionable and unactionable alerts, as measured by accuracy. Precision should be considered in addition to other metrics, like recall, accuracy, and AUC. Studies that report precision, recall, accuracy, and AUC provide a more complete picture of the efficacy of an AAIT.

The remaining studies all used various metrics to evaluate the efficacy of AAIT supplementing ASA. The individual results are summarized Sections 6.8 and 7.12. There are not enough commonalities between the remaining AAIT to provide a cohesive meta-analysis. Overall, the individual results are encouraging that using AAITs to supplement ASA identifies actionable alerts.

9. Limitations

The SLR process was designed to reduce internal validity; however, there are some limitations to the process that introduce bias. The first author primarily handled selection of the studies, with validation efforts by the second author as described in Section 2.3.3. The kappa statistic [18] was utilized to measure the similarity between the two authors ratings, but only a percentage of the studies were considered. Including a search of related work and author's websites probably mitigated missed studies. None of the studies added after Stage 3 via related work and author's website searches were in the original set of papers obtained through the database search. However, improperly selected search terms or inclusion/exclusion criteria may introduce attrition bias.

The quality measurement of the papers is another potential source of internal validity. The first author assessed the quality of the papers based on 10 questions, which may lead to measurement bias. The questions were generated from the authors' experiences with reading and writing research papers and relate to the major sections that are expected to be in research papers.

The first author solely conducted the data extraction portion of the SLR. Therefore, any data is reflective of the author's understanding of the selected study, which is a threat to internal and external validity of the SLR. A list of specific data to extract from each study (as shown in Section 2.4.2, was utilized to minimize the limitation of a single person conducting data extraction.

Table 12
Precision results for seven of the 21 AAIT.

AAIT	Subject	Factor	KLOC	# Alerts	Precision (%)
HISTORYAWARE [55]	Wine	HISTORYAWARE	uk	2860	62
		NAÏVERANKING	uk	2860	53
	Apache	HISTORYAWARE	200	738	42
		NAÏVERANKING	200	738	32
BM08B [6]	TVoM	Positive	91	9740	13
		None	91	9740	7
		Negative	91	9740	5
HW09 [22]	jdom	Average	13	420	89
	Runtime	Average	2	256	98
JKS05 [30]	Linux kernel and classical algorithms	Average	uk	uk	69
HWP [32]	Columba	Maximum	121	2331	17
	Lucene	Maximum	37	1513	25
	Scarab	Maximum	64	1483	67
APM [21]	csvobjects	ATA	1.6	7	32
		CL	1.6	7	50
		ATA + CL	1.6	7	39
	Importscrubber	ATA	1.7	35	34
		CL	1.7	35	20
		ATA + CL	1.7	35	18
	iTrust	ATA	14.1	110	5
		CL	14.1	110	2
		ATA + CL	14.1	110	5
	jbook	ATA	1.3	52	22
		CL	1.3	52	27
		ATA + CL	1.3	52	23
	jdom	ATA	8.4	55	6
		CL	8.4	55	9
		ATA + CL	8.4	55	6
	Runtime	ATA	2.8	98	5
		CL	2.8	98	4
		ATA + CL	2.8	98	3
RPM08 [45]	Google	Screening	uk	1652	78
		All-data	uk	1652	73
		BOW	uk	1652	67
		BOW+	uk	1642	74

10. Conclusions

The goal of this work is to synthesize available research results to inform evidence-based selection of actionable alert identification techniques. We identified 21 studies from literature that propose and evaluate AAITs. The various models proposed in the literature range from the basic alert type selection to machine learning, graph theory, and repository mining. The results generated by the various experiments and case studies vary due to the inconsistent use of evaluation metrics, subject programs, and ASA. Evaluation metrics are required for direct comparison of results. However, for those comparisons to be meaningful, the AAIT should be applied to the same subject programs and ASA. Using a concurrency ASA on a single threaded application would skew the results of an applied AAIT.

Of the results reported in the selected studies, we can observe some high-level trends about AAIT. Seven of the 21 studies reported the precision of their AAIT in classifying or prioritizing alerts. The precision of the AAITs varied, and there is no evidence that the precision of a subject program is associated with the size of the program in terms of KLOC or number of alerts. The AAIT reporting the highest precision for their subject program are the HW09 AAIT [22] and RPM08 AAIT [45]. Both of these AAITs consider the history of the subject program to generate the AAIT and utilize machine learning techniques, which suggest that AAITs that consider the history or that use machine learning may perform better than other models. Evaluation of the HISTORYAWARE [55], ALERTLIFETIME [31], and HWP [32] AAITs provide additional evidence that using the source code repository can identify actionable alerts with a relatively good accuracy and precision.

The selected studies demonstrate that AAITs tend to perform better than a random ordering of alerts or the alerts ordered via the tool's output [21,31,35,36]. This trend across four of the 21 studies suggests that AAITs are an important addition to ASA. The Z-RANKING [36] and FEEDBACK-RANK [35] AAIT show a two to six time improvement over a random baseline at identifying actionable alerts. The ALERTLIFETIME [31] AAIT provides an ordering of important alert types from the project's history that do not match the priority given to the alert's type by the tool. The APM [21] AAIT demonstrated a larger anomaly detection rate curve than the tool ordering of alerts.

Validation of the reported AAITs consisted of an example or set of examples. However, the rigor of the validation depends on the rigor of the baseline of comparison. Some of the studies compared their AAITs against a random baseline, other models, and a benchmark. Still others use as the baseline the output generated by the static analysis tool or in the case of hybrid techniques, the constituent parts were the baseline. In those cases, performance, typically in terms of analysis time or size, was also evaluated. The validation of the remaining studies was a combination of different metrics dependent on the type of AAIT. The current variety of validation techniques preclude a definitive comparison of AAITs reported in literature, but can serve as a starting point for a discussion of a benchmark for validating AAITs.

Both studies that incorporated model checking [44,59] reported promising results. Yu et al. [59] demonstrated that their technique could identify 73 or 74 seeded vulnerabilities and Rungta and Mercer [44] reported the probability of finding an error across 100 trials as 1.0 or close to 1.0. The types of errors detected by each technique differed: Yu et al. focused on finding security vulnerabilities and Rungta and Mercer focused on finding concurrency errors. These results may encourage the use of model checking in combination with other ASA.

ASA is useful for identifying code anomalies, potentially early during software development. The selected studies for the SLR have demonstrated the efficacy of using AAITs to mitigate the costs of unactionable alerts by either classifying alerts as actionable or prioritizing alerts generated by ASA. Developers can focus their alert inspection activities on the alerts they are most likely to want to act on. Ten of the 21 studies use information about the alerts themselves to predict actionable alerts [6,21,22,31,32,34,35,38,40,45]. Six of the selected studies utilize information from the development history of a project to use the past to predict the future [6,22,31,32,45,55]. Twelve of the selected studies use information about the surrounding source code [1,6,7,22,30,36,44,45,55,57–59].

11. Future work

From the analysis of the AAIT studies in literature, we can reflect on RQ4.

- *RQ4*: What are the research challenges and needs in the area of AAITs?

AAITs provide a mechanism for identifying if ASA alerts are actionable outside of the ASA itself. The sophistication of these techniques has increased from a simple alert type selection to hybrid models [8,14], machine learning [22,45], and model checking [44,59]. Nanda et al. [39] suggest that deeper analysis of actionable alerts is required beyond current ASAs. The deeper analysis may complement current ASA through model checking and dynamic analyses that enable a developer to ascertain a concrete error path. Even with a concrete error path, a developer may not want to fix a fault (i.e. the fault is in an exception path, the fix may introduce more faults, etc.); therefore machine learning may be incorporated with these deeper analyses to identify truly actionable alerts.

However, determining the best way to combine current techniques requires that we understand the strengths and weakness of each constituent part. We propose that a comparative evaluation of AAITs is required for further understanding of the strengths and weaknesses of the proposed techniques. There are several requirements for a comparative evaluation: (1) common subject programs (which imply a common programming language for evaluation); (2) common static analysis tools; and (3) a common oracle of actionable and unactionable alerts.

A benchmark for AAIT evaluation would be beneficial to facilitate comparative evaluations of presented and emerging AAIT techniques. Use of a benchmark would allow for the consolidation of results around specific metrics and provide direct comparison by providing subject programs. Sim et al. [47] define a benchmark as providing a motivating comparison, task sample, and evaluation measures. Effective creation of a benchmark requires collaboration among researchers in a specific area, like AAIT [47].

Many of the current AAITs share a motivating comparison in maximizing the precision of actionable alert classification. The task sample and evaluation measures (in addition to precision) are varied. A discussion in the AAIT community could start work toward a benchmark. As a starting point, FAULTBENCH [21] has been proposed to address the need for comparative evaluation of AAITs in the Java programming language. The initial version of FAULTBENCH [21] contained alert oracles (as part of the task sample) that were generated by a person inspecting the alerts in the subject programs. Additional inspections by other researchers could lead to a consensus oracle. Programmatically creating an oracle, as described in Liang et al. [37] work, could reduce the subjectivity. Inspection by other AAIT researchers would improve the benchmark's oracles An ideal benchmark would build upon FAULTBENCH motivating comparison and evaluation metrics to allow for the repeatable, automatic creation of a task sample for any program. The work by

Liang et al. [37] would be a starting point for the repeatable creation of a task sample. By having an automated process, the benchmark could be language independent.

Acknowledgements

We would like to thank the RealSearch reading group, especially Lauren Haywood, for their comments on this paper. This work was funded by the first author's IBM PhD Fellowship.

References

[1] A. Aggarwal, P. Jalote, Integrating static and dynamic analysis for detecting vulnerabilities, in: Proceedings of the 30th Annual International Computer Software and Applications Conference, Chicago, Illinois, USA, September 17–21, 2006, pp. 343–350.

[2] C. Artho, A. Biere, Applying static analysis to large-scale, multi-threaded java programs, in: Proceedings of the 13th Australian Conference on Software Engineering, Canberra, Australia, August 27–28, 2001, pp. 68–75.

[3] R.M. Bell, T.J. Ostrand, E.J. Weyuker, Looking for bugs in all the right places, in: Proceedings of the International Symposium on Software Testing and Analysis, 2006, pp. 61–71.

[4] C. Boogerd, L. Moonen, Prioritizing software inspection results using static profiling, in: Proceedings of the 6th IEEE Workshop on Source Code Analysis and Manipulation, Philadelphia, PA, USA, September 27–29, 2006, pp. 149–160.

[5] C. Boogerd, L. Moonen, Ranking software inspection results using execution likelihood, in: Proceedings of the Philips Software Conference, November, 2006, p. 10.

[6] C. Boogerd, L. Moonen, Assessing the value of coding standards: an empirical study, in: Proceedings of the IEEE International Conference on Software Maintenance, Beijing, China, September 28–October 4, 2008, pp. 277–286.

[7] C. Boogerd, L. Moonen, On the use of data flow analysis in static profiling, in: Proceedings of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, Beijing, China, September 28–29, 2008, pp. 79–88.

[8] P. Chen, H. Han, Y. Wang, S. Shen, X. Yin, B. Mao, L. Xie, INTFINDER: automatically detecting integer bugs in x86 binary program, in: Proceedings of the International Conference on Information and Communications Security, Beijing, China, December, 2009, pp. 336–345.

[9] B. Chess, G. McGraw, Static analysis for security, IEEE Security & Privacy 2 (6) (2004) 76–79.

[10] B. Chess, J. West, Secure Programming with Static Analysis, first ed., Addison-Wesley, Upper Saddle River, NJ, 2007.

[11] J. Cohen, A coefficient of agreement for nominal scales, Educational and Psychological Measurement 20 (1960) 213–220.

[12] C. Csallner, Y. Smaragdakis, JCrasher: an automatic robustness tester for Java, Software – Practice and Experience 34 (11) (2004) 1025–1050.

[13] C. Csallner, Y. Smaragdakis, Check 'n' crash: combining static checking and testing, in: Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, USA, 2005, pp. 422–431.

[14] C. Csallner, Y. Smaragdakis, T. Xie, DSD-crasher: a hybrid analysis tool for bug finding, ACM Transactions on Software Engineering and Methodology 17 (2) (2008) 1–36.

[15] D. Engler, B. Chelf, A. Chou, S. Hallem, Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions, in Operating Systems Design and Implementation, San Diego, CA, 2000.

[16] M.D. Ernst, J. Cockrell, W.G. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, IEEE Transactions on Software Engineering 27 (2) (2001) 99–123.

[17] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, R. Stata, Extended static checking for java, in: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany, 2002, pp. 234–245.

[18] J.L. Fleiss, J. Cohen, B.S. Everitt, Large sample standard errors of kappa and weighted kappa, Psychological Bulletin 72 (1969) 323–327.

[19] E. Haugh, M. Bishop, Testing C programs for buffer overflow vulnerabilities, in: Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, February, 2003, pp. 123–130.

[20] S. Heckman, L. Williams, A measurement framework of alert characteristics for false positive mitigation models, in: North Carolina State University TR-2008-23, October 6, 2008.

[21] S. Heckman, L. Williams, On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques, in: Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement, Kaiserslautern, Germany, October 9–10, 2008, pp. 41–50.

[22] S. Heckman, L. Williams, A model building process for identifying actionable static analysis alerts, in: Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation, Denver, CO, USA, 2009, pp. 161–170.

[23] S.S. Heckman, A Systematic Model Building Process for Predicting Actionable Static Analysis Alerts, Dissertation, Computer Science, North Carolina State University, 2009.

[24] G.J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2003.

[25] D. Hovemeyer, W. Pugh, Finding bugs is easy, in: Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, British Columbia, Canada, October 24–28, 2004, pp. 132–136.

[26] IEEE, IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, in no., 1990.

[27] IEEE, IEEE 1028-1997 (R2002) IEEE Standard for Software Reviews, no., 2002.

[28] IEEE, ISO/IEC 24765:2008 Systems and Software Engineering Vocabulary, 2008.

[29] S.C. Johnson, Lint, a C Program Checker, Bell Laboratories, Murray Hill, NJ, 1978.

[30] Y. Jung, J. Kim, J. Shin, K. Yi, Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis, in: Proceedings of the 12th International Static Analysis Symposium, Imperial College London, UK, 2005, pp. 203–217.

[31] S. Kim, M.D. Ernst, Prioritizing warning categories by analyzing software history, in: Proceedings of the International Workshop on Mining Software Repositories, Minneapolis, MN, USA, May 19–20, 2007, p. 27.

[32] S. Kim, M.D. Ernst, Which warnings should I fix first?, in: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, September 3–7, 2007, pp. 45–54.

[33] B. Kitchenham, Procedures for Performing Systematic Reviews, Joint Technical Report, Keele University Technical Report (TR/SE-0401) and NICTA Technical Report (0400011T.1) July 2004, 2004.

[34] D. Kong, Q. Zheng, C. Chen, J. Shuai, M. Zhu, "ISA: a source code static vulnerability detection system based on data fusion, in: Proceedings of the 2nd International Conference on Scalable Information Systems, Suzhou, China, June 6–8, 2007, p. 55.

[35] T. Kremenek, K. Ashcraft, J. Yang, D. Engler, Correlation exploitation in error ranking, in: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Newport Beach, CA, USA, 2004, pp. 83–93.

[36] T. Kremenek, D. Engler, Z-ranking: using statistical analysis to counter the impact of static analysis approximations, in: Proceedings of the 10th International Static Analysis Symposium, San Diego, California, 2003, pp. 295–315.

[37] G. Liang, L. Wu, Q. Wu, Q. Wang, T. Xie, H. Mei, Automatic construction of an effective training set for prioritizing static analysis warnings, in: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September, 2010, pp. 93–102.

[38] N. Meng, Q. Wang, Q. Wu, H. Mei, An approach to merge results of multiple static analysis tools (short paper), in: Proceedings of the Eight International Conference on Quality Software, Oxford, UK, August 12–13, 2008, pp. 169–174.

[39] M.G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, P. Balachandran, Making defect-finding tools work for you, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Cape Town, South Africa, May 2–8, vol. 2, 2010, pp. 99–108.

[40] H. Ogasawara, M. Aizawa, A. Yamada, Experiences with program static analysis, in: Proceedings of the 1998 Software Metrics Symposium, Bethesda, MD, USA, November 20–21, 1998, pp. 109–112.

[41] T.J. Ostrand, E.J. Weyuker, R.M. Bell, Where the bugs are, in: Proceedings of the International Symposium on Software Testing and Analysis, 2004, pp. 86–96.

[42] C. Pacheco, M.D. Ernst, Eclat: automatic generation and classification of test inputs, in: Proceedings of the 19th European Conference on Object-Oriented Programming, Glasgow, Scotland, July 27–29, 2005, pp. 504–527.

[43] G. Rothermel, R.H. Untch, C. Chu, M.J. Harrold, Prioritizing test cases for regression testing, IEEE Transactions on Software Engineering 27 (10) (2001) 929–948.

[44] N. Rungta, E.G. Mercer, A meta heuristic for effectively detecting concurrency errors, in: Proceedings of the 4th International Haifa Verification Conference on Hardware and Software: Verification and Testing, Haifa, Israel, October, 2008, pp. 23–37.

[45] J.R. Ruthruff, J. Penix, J.D. Morgenthaler, S. Elbaum, G. Rothermel, Predicting accurate and actionable static analysis warnings: an experimental approach, in: Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, May 10–18, 2008, pp. 341–350.

[46] M. Shaw, Writing good software engineering research papers: minitutorial, in: Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, USA, May 3–10, 2003, pp. 726–736.

[47] S.E. Sim, S. Easterbrook, R.C. Holt, Using benchmarking to advance research: a challenge to software engineering, in: Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, USA, May 3–10, 2003, pp. 74–83.

[48] W.F. Tichy, Should computer scientists experiment more?, Computer 31 (5) (1998) 32–40.

[49] S.B. Vardeman, J.M. Jobe, Basic Engineering Data Collection and Analysis, first ed., Duxbury, Pacific Grove, CA, 2001.

[50] J. Viega, J.T. Floch, Y. Kohno, G. McGraw, "ITS4: a static vulnerability scanner for C and C++ code, in: Proceedings of the 16th Annual Conference on Computer Security Applications, New Orleans, LA, USA, December 11–15, 2000, pp. 257–267.

[51] W. Visser, K. Havelund, G. Brat, S. Park, Model checking programs, in: Proceedings of the 15th IEEE International Conference on Automated Software Engineering, Grenoble, France, September, 2000, pp. 3–11.
[52] D. Wagner, S.F. Jeffrey, E.A. Brewer, A. Aiken, A first step towards automated detection of buffer overrun vulnerabilities, in: Proceedings of the Network and Distributed Systems Security Conference, San Diego, CA, USA, February 2–4, 2000, 2000, pp. 3–17.
[53] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, M. Schwalb, An evaluation of two bug pattern tools for java, in: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, Lillehammer, Norway, April 9–11, 2008, pp. 248–257.
[54] D.W. Wall, Predicting program behavior using real or estimated profiles, in: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, June 26–28, 1991, pp. 59–70.
[55] C.C. Williams, J.K. Hollingsworth, Automatic mining of source code repositories to improve bug finding techniques, IEEE Transactions on Software Engineering 31 (6) (2005) 466–480.
[56] I.H. Witten, E. Frank, Data Mining: Practical Machine Learning Tools and Techniques, second ed., Morgan Kaufman, Amsterdam, 2005.
[57] S. Xiao, C. Pham, Performing high efficiency source code static analysis with intelligent extensions, in: Proceedings of the 11th Asia-Pacific Software Engineering Conference, Busan, Korea, November 30–December 3, 2004, pp. 346–355.
[58] K. Yi, H. Choi, J. Kim, Y. Kim, An empirical study on classification methods for alarms from a bug-finding static C analyzer, Information Processing Letters 102 (2–3) (2007) 118–123.
[59] L. Yu, J. Zhou, Y. Yi, J. Fan, Q. Wang, A hybrid approach to detecting security defects in programs, in: Proceedings of the 9th International Conference on Quality Software, Jeju, Korea, August 24–25, 2009, pp. 1–10.
[60] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, M. Vouk, On the value of static analysis for fault detection in software, IEEE Transactions on Software Engineering 32 (4) (2006) 240–253.
[61] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects in eclipse, in: Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering, Minneapolis, MN, USA, May 20, 2007, p. 9.