A Systematic Literature Review of Actionable Alert Identification Techniques for Automated Static Code Analysis

Sarah Heckman (corresponding author) and Laurie Williams

North Carolina State University
890 Oval Drive, Campus Box 8206, Raleigh, NC 27695-8206
(phone) +1.919.515.2042
(fax) +1.919.515.7896
[heckman, williams]@csc.ncsu.edu

Abstract

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 eighteen peer-reviewed studies of AAITs. The techniques use alert type selection; contextual information; data fusion; graph theory; machine learning; mathematical and statistical models; or test case failures to classify and prioritize actionable alerts. All of the AAITs are evaluated via an experiment or case study 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 eighteen 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 analysis language in the selected studies preclude meta-analysis and prevent the current results from informing evidenced-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.

Keywords: automated static analysis, systematic literature review, actionable alert identification, unactionable alert mitigation, warning prioritization, actionable alert prediction

1. Introduction

Static analysis is "the process of evaluating a system or component based on its form, structure, content, or documentation" [20]. Automated static analysis (ASA) can identify common coding problems

early in the development process via a tool that automates the inspection[1] of source code [47]. ASA reports potential source code anomalies[2], which we call *alerts*, like null pointer dereferences, buffer overflows, and style inconsistencies [19]. 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* [16-17, 35]. When an alert is not an indication of an actual code anomaly or the alert is deemed unimportant 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* [16-17].

Static analysis tools generate many alerts; an alert density of 40 alerts per thousand lines of code (KLOC) has been empirically observed [16]. Developers and researchers found that 35% to 91% of reported alerts are unactionable [1, 3, 16-17, 24-25, 28-29]. 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 [3, 25, 28-29]. Suppose, a tool reports 1000 alerts and each alert requires five minutes for inspection. The time to inspect the alerts would take 10.4 uninterrupted eight-hour 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 [41].

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 [6-7]. Additionally, the development of algorithms underlying ASA requires a trade-off between the level of analysis and execution time [6]. 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 [46].

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. *Classification* AAITs divide alerts into two groups: alerts predicted to be actionable and alerts predicted to be unactionable [16]. *Prioritization* AAITs order alerts by the likelihood an alert is an indication of an actionable alert [16].

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" [26]. The specific objectives of this SLR are the following:

- To identify categories of AAIT input artifacts;
- To summarize current research solutions for AAIT;
- To synthesize the current results from AAIT; and
- To identify the research challenges and needs in the area of AAIT.

---

[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" [22].

[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" [21].

[3] For this research, we consider the generation of an alert as the indication of a potential anomaly. Therefore, sound static analysis implies that all reported alerts are anomalies (actionable) [9].

[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 [9].

[5] AAITs are techniques that identify actionable alerts. Some AAITs have been referred to as false positive or unactionable alert mitigation [16-17], warning prioritization [24-25], and actionable alert prediction [35].

The remainder of this paper is as follows: Section 2 reports the SLR method followed by an overview of the selected study's 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 [26] 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 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?

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.

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.

Table 1: Keywords describing static analysis alerts and AAITs

Alert Descriptor	Identification Technique
alert	prioritization
defect	ranking
fault	classification
bug	reduction
warning	detection
	prediction

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

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.

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) elimination of studies based upon reading the abstract; and 3) further elimination 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 6320 distinct papers from the database search and selected 466 that moved to the next stage of papers selection. A 467th paper was added to Stage 2 of paper selection from the verification efforts described in Section 2.3.3. At Stage 3, 34 from the original search had relevant abstracts and warranted further reading. The verification efforts and reading of the paper's related work sections added another seven papers for 41 papers. The final selected studies consisted of 16 papers and an additional two, new, papers found by searching the authors' websites for later works, for a total of 18 selected studies.

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

Stage	Papers	Added Papers	Total Papers
Stage 1: by title	6320	0	6320
Stage 2: by abstract	466	1	467
Stage 3: by paper	34	7	41
Final: selected studies	16	2	18

Studies selected at each stage of the selection process met our inclusion criteria:

- Full and short peer reviewed papers with empirical results;
- Post ASA run alert classification or prioritization; and
- Focus on 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;
- Dynamic analyses; and

- 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).

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.

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.

The website of the selected papers' authors' of post Stage 3 papers were mined for additional studies that may have been missed during the database search. As a result of this search, an additional two studies were identified for inclusion in the final set of 18, one of which replaced an earlier study by the same authors due to the reporting of earlier results in addition to new results. A third study, recently published of the SLR authors' work, was also included in the final set.

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 the following information:

- Title
- Author(s)
- Conference (abbreviation and full name)
- Publication Month
- Publication Year
- Abstract

Additionally, at each phase of 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, 300 (4.7%) of the studies were randomly selected for the second author to evaluate. The first author prepared the selection, ensuring that the sample had the same proportion of selected and rejected studies as the full population. Two hundred and eighty seven (95.6%) 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. Therefore, the level of error in the selection process is within an acceptable range.

After Stage 2, the second author evaluated the abstracts for 50 (10.7%) randomly selected studies. We again ensured that the sample had the same distribution of selected and rejected studies as the full population of studies. Forty-five (90%) 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. Again, the level of error in the selection process is within an acceptable range. The two authors discussed all papers selected for the final set of studies by the first author for inclusion in the SLR.

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.

Table 3: Quality assessment questions

1.	Is there a clearly stated research goal related to the identification of actionable alerts?
2.	Is there a defined and repeatable AAIT?
3.	Are the limitations to the AAIT enumerated?
4.	Is there a clear methodology for validating the AAIT?
5.	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?
6.	Is there a control technique or process (random, comparison)?
7.	Are the validation metrics relevant (e.g. evaluate the effectiveness of the AAIT) to the research objective?
8.	Were the presented results clear and relevant to the research objective stated in the study?
9.	Are the limitations to the validation technique enumerated?
10.	Is there a listing of contributions from the research?

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 Goal
- AAIT Type
- AAIT
- AAIT Limitations
- Artifact Characteristics
- Validation Methodology (experiment, case study, etc.)
- Validation Subjects
- Evaluation Metrics and Definitions

- Evaluation Results
- Static Analysis Tools Used
- Study Limitations

All of the SLR data collected from the selected studies, except the results from the paper, were maintained in an internal wiki, with one wiki page per paper. Each of the data extraction categories, listed above, were filled in with the appropriate text directly from the paper with occasional changes to a consistent vocabulary. 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. An AAIT is run on a subject program or application for experimental evaluation of the AAIT. The results spreadsheet was used to determine if there are any trends in evaluating AAITs.

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 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 18 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.

Table 4: Publication year

Year	**#**
1998	1
2003	1
2004	2
2005	2
2006	1
2007	4
2008	6
2009 (before May)	1
Total	**18**

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.

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 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

Study quality ranged from 3 to 10, where a quality value of 10 is highest, as measured via the ten 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 8.2, 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 1-15, with an average of four subject programs per study.

Table 6: Summarization of the selected studies

AAIT Name	Study	Study Quality	Artifact Char. Category	ASA	Lang.	Dis. in Sec.	AAIT Approach	Evaluation Methodology
AJ06	Aggarwal and Jalote, 2006 [1]	6	CC	BOON	C	6.4	contextual information	other
ALERTLIFTIME	Kim and Ernst, 2007a [24]	8.5	AC, SCR	FINDBUGS, PMD, JLINT	Java	7.4	math/stat models	other baseline comparison
APM	Heckman and Williams, 2008 [16]	10	AC	FINDBUGS	Java	7.9	math/stat models	benchmark
BM08B	Boogerd and Moonen, 2008b [4]	9	AC, CC, SCR, BDB	QA C, QMORE	C	6.5	graph theory	other
CHECK 'N' CRASH, DSD-CRASHER	Csallner, et al., 2008 [10]	10	DA	ESC/JAVA	Java	6.6	test case failures	other model comparison
ELAN, $EFAN_H$, $EFAN_V$	Boogerd and Moonen, 2008a [5]	6.5	CC	UNSPEC.	C	7.8	graph theory	other baseline comparison
FEEDBACK-RANK	Kremenek, et al., 2004 [28]	8	AC	MC	C	7.2	machine learning	random and optimal comp.
HISTORY-AWARE	Williams and Hollingsworth, 2005 [43]	10	CC, SCR	RETURN VALUE CHECKER	C	6.3	math/stat models	other model comparison
HW09	Heckman and Williams, 2009 [17]	10	AC, CC, SCR	FINDBUGS	Java	6.7	machine learning	benchmark
HWP	Kim and Ernst, 2007b [25]	8	AC, SCR	FINDBUGS, PMD, JLIINT	Java	7.5	math/stat models	train and test
ISA	Kong, et al., 2007 [27]	8	AC	RATS, ITS4, FLAWFINDER	C	7.6	data fusion	other model comparison
JKS05	Jung, et al., 2005 [23]	8	CC	AIRIC	C	7.3	machine learning	train and test
MMW08	Meng, et al., 2008 [30]	3	AC	FINDBUGS, PMD, JLINT	Java	7.10	data fusion	other
OAY98	Ogasawara, et al., 1998 [31]	7	AC	QA C	C	6.1	alert type selection	other
RPM08	Ruthruff, et al., 2008 [35]	10	AC, CC, SCR	FINDBUGS	Java	7.11	math/stat models	other model comparison
SCAS	Xiao and Pham, 2004 [45]	7	CC	UNSPEC.	C	6.2	contextual information	other
YCK07	Yi, et al., 2007 [46]	9	CC	AIRIC	C	7.7	machine learning	train and test
Z-RANKING	Kremenek and Engler, 2003 [29]	10	CC	MC	C	7.1	math/stat models	random and optimal comp.

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 [15] 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. 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].

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.

Table 7: Studies by category of artifact characteristics

Artifact Characteristics Category	Number	Percent
Alert characteristics (AC)	10	56%
Code characteristic (CC)	10	56%
Source code repository metrics (SCR)	6	33%
Bug database metrics (BDB)	1	6%
Dynamic analysis metrics (DA)	1	6%

Of particular note, 56% percent of the selected studies used ACs as an independent variable. Nine of the 10 studies that used ACs used the alert's type to predict actionable alerts. An alert's type is an identifier generated by the ASA tool to describe the alert generated.

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 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

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 and unactionable alerts. Additionally, the approach taken to use the artifact characteristics to prediction 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 18 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.

Table 9: AAIT Approach

AAIT Approach	Number
Alert type selection	1
Contextual Information	2
Data Fusion	2
Graph Theory	2
Machine Learning	4
Mathematical and Statistical Models	6
Test Case Failures	1

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) [24-25, 28]. 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]. ASA may also not understand code constructs like pointers, which may lead to a large number of unactionable alerts [1, 45]. 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 [27].

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 [3, 5]. 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 [4].

5.1.5. Machine Learning

Machine learning "is the extraction of implicit, previously unknown, and potentially useful information about data" [44]. 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 [17, 23, 28, 46].

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 [29]. Other AAITs may use the history of the code to build a linear model that may predict actionable alerts [24-25, 35, 43]. Additionally, knowledge about the ASA tools and the observed relationships between the alerts can be used to create mathematical models [16].

5.1.7. Test Case Failures

Unlike static analysis, the results of dynamic analyses do not require inspection because a failing condition is identified through program execution [10]. 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 [10]. 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 [10].

5.2. AAIT Evaluation Methodologies

In the selected studies, all of the alert prioritization techniques were evaluated using an experiment or case study 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.

Table 10: Evaluation Methodologies for Selected Studies

Evaluation Methodologies	Number
Baseline Comparison	2
Benchmarks	2
Comparison to Other AAIT	4
Random and Optimal Comparison	2
Train and Test	3
Other	5

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 [5, 24] 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 [24]. 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 [5].

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 [37]. 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" [20]. 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 [36]. 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 [36]. Additionally, successful benchmarks provide a vehicle for collaboration and commonality in the research community [36].

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 [28-29] 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 [16, 28-29]. A random ranking is the random selection of alerts without replacement [16, 28-29]. 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" [28].

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 [44]. For models that use the project history, the training data may come from the first *i* revisions of source code [17, 25]. The test data then is the remaining *i*+1 to *n* revisions [17, 25]. 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 [46].

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 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 12, in Appendix A, 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 18 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 [23, 30].

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 12 in Appendix A. 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) [4, 16-17, 31, 48]:** classifying an alert as actionable when the alert is actionable.
- **True negative classification (TN) [16-17, 48]:** classifying an alert as unactionable when the alert is unactionable.
- **False positive classification (FP) [16-17, 27, 48]:** classifying an alert as actionable when the alert is actually unactionable.
- **False negative classification (FN) [16-17, 27, 48]:** 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. Figure 1 is a classification table that model the metrics discussed above.

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

Figure 1: Classification Table (adapted from Zimmermann, et al. [48])

The following metrics evaluate the classification of static analysis alerts:

- **Precision [1, 16-17, 25, 43-44]**: the proportion of correctly classified anomalies (TP) out of all alerts predicted as anomalies (TP + FP). The precision calculation is presented in Equation 1.

  $$precision = (TP) / (TP + FP) \quad (1)$$

- **Recall (also called True Positive Rate or Sensitivity) [16-17, 43-44]**: the proportion of correctly classified anomalies (TP) out of all possible anomalies (TP + FN). The recall calculation is presented in Equation 2.

  $$recall = (TP) / (TP + FN) \quad (2)$$

- **Accuracy [5, 16-17, 44]**: the proportion of correct classifications (TP + TN) out of all classifications (TP + TN + FP + FN). The accuracy calculation is presented in Equation 3.

  $$accuracy = (TP + TN) / (TP + TN + FP + FN) \quad (3)$$

- **False Positive Rate [43-44]:** 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 Equation 4.

  $$false\ positive\ rate = (FP) / (FP + TN) \quad (4)$$

- **Number of Test Cases Generated [1, 10]:** the number of automated test cases generated by hybrid techniques that generate test cases.

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 [16]:** 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 [42]:** measures how closely the alerts prioritized by the AAITs match the actual program executions.

- ***r* [4, 24, 38]:** the correlation coefficient is a measure of the strength of association between independent and dependent variables.

- **Chi-square test [43]:** comparison of false positive rates to see if the use of an AAIT produces a statistically significant reduction.

- **Area Under the Curve (AUC) [16, 44, 46]:** 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 [44, 46] and anomaly detection rate curve [16]. 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 [28-29]:** the number of alert inspections required to identify all actionable alerts.

- **Prioritization Technique's Improvement over Random [28-29]:** 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 [16]. The subsections below describe the seven out of 18 AAITs from the selected studies that are classification AAITs.

For each AAIT, we provide 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., 1998
- Artifact Characteristics: AC
- ASA: QA C[6]
- Language: C
- AAIT Type: Alert Type Selection (5.1.1)
- Research Methodology: Other (5.2.6)

Ogasawara, et al. [31] present a method for using ASA during development whereby only the alert types that identify the most actionable alerts are used. 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.

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.

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] QA C is a static analysis tool developed by Programming Research (http://www.programmingresearch.com/QAC_MAIN.html).

6.2. SCAS

- Study: Xiao and Pham, 2004
- Artifact Characteristics: CC
- ASA: unspecified
- Language: C
- AAIT Type: Contextual Information (5.1.2)
- Research Methodology: Other (5.2.6)

Xiao and Pham [45] use contextual information about the code under analysis to extend a static analysis tool. 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. The additional techniques suggested may generate overhead making them too costly to use in-process. The SCAS AAIT suppresses 32% of the generated alerts.

6.3. HISTORYAWARE

- Study: Williams and Hollingsworth, 2005
- 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)

Williams and Hollingsworth [43] use source code repository mining to drive the creation of an ASA tool and to improve the prioritized listing of alerts. Adding a return value check on a function call was a common bug fix in the Apache httpd[7] 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.

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. The precision and recall of the classifications are measured. The precision of the top 50 alerts generated by static analysis is 62.0% for HISTORYAWARE and 53.0% for NAÏVERANKING for Wine[8] and 42.0% for HISTORYAWARE and 32.0% for NAÏVERANKING for

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

[8] 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/.

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, 2006
- Artifact Characteristics: CC
- ASA: BOON [40]
- Dynamic Analysis: STOBO [14]
- Language: C
- AAIT Type: Contextual Information (5.1.2)
- Research Methodology: Other (5.2.6)

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. The ASA tool, BOON [40], has difficulty in understanding aliased pointers, which may lead to BOON missing buffer overflow vulnerabilities. Dynamic tools, like STOBO [14], 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.

The study of the AJ09 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.

Running AJ09 on rdesktop[9] and wzdftpd[10] lead to an increase in ASA accuracy and a reduction of test cases required to find buffer overflow vulnerabilities. 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, AJ09, 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, 2008b
- 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)

Boogerd and Moonen [4] 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. 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

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

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

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

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. For each alert type, the actionable alert rate was calculated. 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.

6.6. CHECK 'N' CRASH and DSD-CRASHER

- Study: Csallner, et al., 2008
- Artifact Characteristics: DA
- ASA: ESC/JAVA [13]
- Dynamic Analysis: DAIKON [12] & JCRASHER [8]
- Language: Java
- AAIT Type: Test Case Failures (5.1.4)
- Research Methodology: Other Model Comparison (5.2.2)

The CHECK 'N' CRASH and DSD-CRASHER tooling are evaluated in Csallner, et al. [10]. We only include these two tools because of the summarization of the results when using a hybrid AAIT. JBoss JMS[11] and Groovy[12] are the subject programs for comparing CHECK 'N' CRASH with DSD-CRASHER. 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 [33] 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 [13], finds bugs seeded in open source projects. The experiment considered three versions of Apache XML Security[13] 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, 2009
- Artifact Characteristics: AC, CC, SCR
- ASA: FINDBUGS [19]
- Language: Java
- AAIT Type: Machine Learning (5.1.6)
- Research Methodology: Benchmark (5.2.3)

---

[11] JBoss JMS is the messaging service component of the JBoss J2EE application server. JBoss may be found at: http://www.jboss.org.
[12] Groovy is a dynamic programming language for the Java Virtual Machine. Groovy may be found at: http://groovy.codehaus.org/.
[13] Information about Apache's XML Security module may be found at: http://santuario.apache.org/.

Heckman and Williams [17-18] 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. 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. The limitations to this process concern choosing the appropriate artifact characteristics and models for the classification of static analysis alerts.

The machine learning-based model building process is evaluated on two FAULTBENCH [16] subject programs. The Weka [44] machine learning tool is used to generate artifact characteristic sets and classification models. The models are generated and evaluated by using ten ten-fold 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 generated for two of the FAULTBENCH subject programs: jdom[14] and runtime[15]. 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. Classification Results Discussion

Ogasawara, et al. [31] 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 [10], and SCAS [45] 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. The AAIT proposed by Aggarwal, et al. [1] identifies places in the code where dynamic analysis 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.

AAITs proposed by Boogerd and Moonen [4], Heckman and Williams [17], and Williams and Hollingsworth [43] 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 5.0-13.0%, 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 11.

---

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

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

Table 11: Precision results for seven of the 18 AAIT

AAIT	Subject	Factor	KLOC	# Alerts	Precision
HISTORYAWARE	Wine	HISTORYAWARE	uk	2860	62%
		NAÏVERANKING	uk	2860	53%
	Apache	HISTORYAWARE	200	738	42%
		NAÏVERANKING	200	738	32%
BM08B	TVoM	POSITIVE	91	9740	13%
		NONE	91	9740	7%
		NEGATIVE	91	9740	5%
HW09	jdom	AVERAGE	13	420	89%
	runtime	AVERAGE	2	256	98%
JKS05	Linux kernel and classical algorithms	AVERAGE	uk	uk	69%
HWP	Columba	MAXIMUM	121	2331	17%
	Lucene	MAXIMUM	37	1513	25%
	Scarab	MAXIMUM	64	1483	67%
APM	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	Google	SCREENING	uk	1652	78%
		ALL-DATA	uk	1652	73%
		BOW	uk	1652	67%
		BOW+	uk	1642	74%

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 [16]. 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, 2003
- Artifact Characteristics: CC

- ASA: MC [11]
- Language: C
- AAIT Type: Mathematical and Statistical Models (5.1.5)
- Research Methodology: Random and Optimal Comparison (5.2.1)

Kremenek and Engler [29] proposed a statistical model for prioritizing static analysis alerts. Unlike most other ASA, the MC [11] 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]" [29]. 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. A limitation of the Z-RANKING technique is that the prioritization's success depends on the grouping operator.

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 detectors in the MC ASA system: lock error detectors, free error detectors, and string format error detectors. The cumulative number of bugs discovered is plotted on a graph for each inspection. 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 x 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., 2004
- Artifact Characteristics: AC
- ASA: MC [11]
- Language: C
- AAIT Type: Machine Learning (5.1.6)
- Research Methodology: Random and Optimal Comparison (5.2.1)

Based on the intuition that alerts sharing an artifact characteristic tend to be either all actionable or all unactionable, Kremenek et al. [28] developed an adaptive prioritization algorithm, FEEDBACK-RANK. 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.

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.

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." The results show that all detectors in the MC system show a 2-8x improvement of performance ratio over random when using FEEDBACK-RANK. The self-trained model for the ASA tool, Alock, showed a 6-8x improvement of performance ratio over random when seeded with partial knowledge of some alert classifications.

7.3. JKS05

- Study: Jung, et al., 2005
- Artifact Characteristics: CC
- ASA: AIRIC [23]
- Language: C
- AAIT Type: Machine Learning (5.1.6)
- Research Methodology: Train and Test (5.2.5)

Jung, et al. [23] use a Bayesian network to generate the probability of an actionable alert 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.

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 were "… some parts of the Linux kernel and programs that demonstrate classical algorithms." 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
- Artifact Characteristics: AC and SRC
- ASA: FINDBUGS [19], PMD[16], JLINT[17]
- Language: Java
- AAIT Type: Mathematical and Statistical Models (5.1.5)
- Research Methodology: Other Baseline Comparison (5.2.4)

---

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

[17] JLINT is ASA for Java: http://artho.com/jlint/

Kim and Ernst [24] prioritize alert types by the average lifetime of alerts sharing the type. 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. 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 [24].

Validation of the ALERTLIFETIME AAIT compared the alerts ordered by lifetime with alerts ordered by the tool specified severity. 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
- Artifact Characteristics: AC and SCR
- ASA: FINDBUGS [19], PMD, JLINT
- Language: Java
- AAIT Type: Mathematical and Statistical Model (5.1.5)
- Research Methodology: Other Baseline Comparison (5.2.4)

Kim and Ernst [25] use the commit messages and code changes in the source code repository to prioritize alert types. 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- α. 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. 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 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. The precision of the tool's alert prioritization with the prioritization of alerts based on the project's history were compared. The best precision for the three subject programs (Columba, Lucene, and Scarab) 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., 2007
- Artifact Characteristic: AC
- ASA: RATS[18], ITS4 [39], FLAWFINDER[19]
- Language: C
- AAIT Type: Data Fusion (5.1.7)

---

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

- Research Methodology: Other Model Comparison (5.2.2)

Kong, et al. [27] use data fusion to identify vulnerable code using alerts generated by ASAs focused on finding security vulnerabilities. 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. The technique is limited by the mapping of alerts between tools.

All of the subject programs used by Kong, et al., listed in Appendix A in Table 12, have known vulnerabilities, which provide a measure of how well ISA and the individual ASA tools perform. The prioritization of the ISA AAIT is compared with the prioritization of the individual ASA tools that make up the ISA tool. 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., 2007
- Artifact Characteristics: CC
- ASA: AIRIC [23]
- Language: C
- AAIT Type: Machine Learning (5.1.6)
- Research Methodology: Train and Test (5.2.5)

Yi, et al. [46] compare the classification of actionable and unactionable alerts for several machine learning algorithms. 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[20] was used to train and test the models.

The YCK07 models were evaluated on 36 files and 22 programs, the details of which are not provided. 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. The AUC for the ROC curves varied from 0.87-0.93. 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, $EFAN_H$, $EFAN_V$

- 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)

---

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

Boogerd and Moonen [5] prioritize alerts by execution likelihood [3] and by execution frequency [5]. Execution likelihood is defined as "the probability that a given program point will be executed at least once in an arbitrary program run" [5]. Execution frequency is defined as "the average frequency of [program point] *v* over all possible distinct runs of [program] *p*" [5]. 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 [5].

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 [5]. The ELAN AAIT (introduced in [3]) 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.

Five open source programs (Antiword, Chktex, Lame, Link, and Uni2Ascii) were used in the case study to compare ELAN and EFAN. 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.

Wall's unweighted matching method [42] compares the prioritized list of alerts with the list of alerts ordered by the actual execution data and produces a measure of correlation. 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 [19]
- Language: Java
- AAIT Type: Mathematical and Statistical Models (5.1.5)
- Research Methodology: Benchmarks (5.2.3)

Heckman and Williams [16] adaptively prioritize individual alerts using the alert's type and location in the source code. 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 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. An assumption of the model is that alerts sharing an alert type or code location are likely to be all actionable or all unactionable. 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).

Three versions of APM were evaluated on the six subject programs of the FAULTBENCH benchmark: csvobject, importscrubber, iTrust, jbook, jdom, and org.eclipse.core.runtime. The three prioritization models were compared with the tool's ordering of alerts and with each other. 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.

[34]. 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 to 0.8) at a statistically significant level for five of the six subject programs. The average accuracy for the three APM models ranged from 67-76% 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., 2008
- Artifact Characteristics: AC
- ASA: FINDBUGS [19], PMD, JLINT
- Language: Java
- AAIT Type: Data Fusion (5.1.7)
- Research Methodology: Other (5.2.6)

Meng et al. [30] propose an approach that merges alerts that are common across multiple static analysis tools run on the same source code. 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. The MMW08 technique has the same limitations as Kong et al. [27].

Evaluation of the MMW08 technique was on a small, unnamed, subject program, which is not listed in Table 12. They run FindBugs, PMD, and JLint on the subject program. Meng et al. [30] 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., 2008
- Artifact Characteristics: AC, CC, SCR
- ASA: FINDBUGS [19]
- Language: Java
- AAIT Type: Mathematical and Statistical Models (5.1.5)
- Research Methodology: Other Model Comparison (5.2.2)

Ruthruff et al. [35] use a logistic regression model to predict actionable alerts. 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.

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 [2, 32]. Ruthruff et al. [35] 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.

The cost in terms of time to generate the data and build the models was compared in addition to the precision of the predictions. The APM08 model building technique, which used screening, took slightly less than seven hours to build and run, which is reasonable compared to the five 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%-86.6%, which was higher than the BOW and BOW+ models with precision between 60.9%-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 [3, 5] 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 [29] 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 [28] 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. [28-29] provide empirical evidence for the efficacy of prioritization techniques; however, other prioritization techniques did not report the same metrics, precluding comparison.

Jung, et al. [23], Kim and Ernst [25], Yi, et al. [46], Heckman and Williams [16], and Ruthruff, et al. [35] report one or more classification (5.4) and/or prioritization metrics (5.5), but specifically all five studies 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 11 summarizes the five studies that report precision. Ruthruff, et al. [35] demonstrate the highest precision with their RPM08 AAIT. Heckman and Williams' [16] 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 [24] 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. [46] 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. [27] 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. [30] 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 18 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 11 presents the combined precision data from seven of the classification and prioritization AAITs, where data are available.

The precisions reported in Table 11 varied from 3%-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 [16]. The other AAIT calculated precision from the initial classification or prioritization.

Figures 2 and 3, from the data in Table 11, 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 [38]. 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 [17] 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. [35] 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.

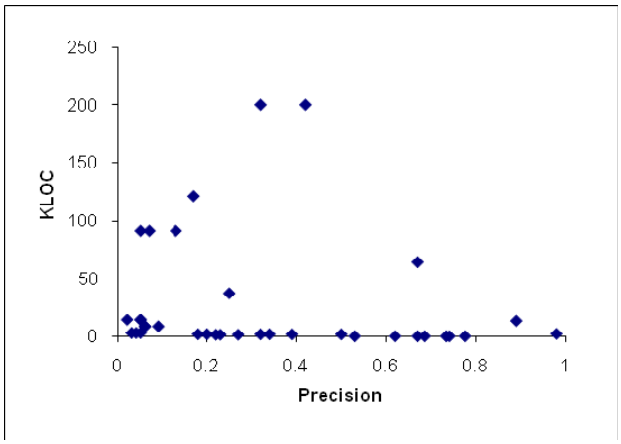

Figure 2: The precision reported for each of the subject-factor pairs for the seven studies that reported precision against KLOC.

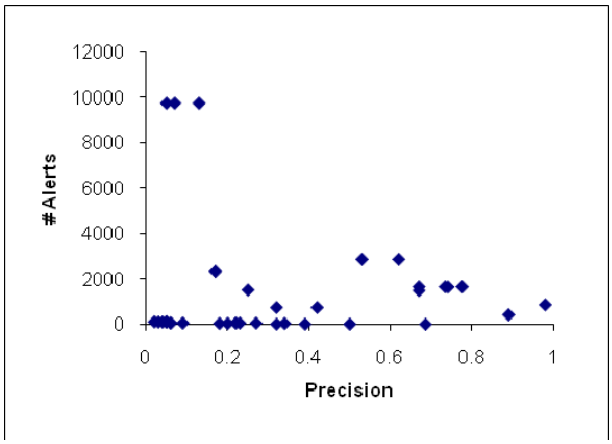

Figure 3: The precision reported for each of the subject-factor pairs for the seven studies that reported precision against the number of alerts.

BMO8B and APM showed the lowest precision during evaluation. Precision is low when the actionable alert predictions are incorrect and there are many more FPs than TPs. Boogerd and Moonen [4] 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 [16] 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 11 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. Conclusions

The goal of this work is to synthesize available research results to inform evidence-based selection of actionable alert identification techniques. We identified 18 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. Seven of the 18 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 [17] and RPM08 AAIT [35]. Both of these AAITs consider the history of the subject program to generate the AAIT, which suggest that AAITs that consider the history may perform better than other models. Evaluation of the HISTORYAWARE [43], ALERTLIFETIME [24], and HWP [25] 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 a trend that AAITs tend to perform better than a random ordering of alerts or the alerts ordered via the tool's output [16, 24, 28-29]. This trend across four of the 18 studies suggests that AAITs are an important addition to ASA. The Z-RANKING [29] and FEEDBACK-RANK [28] AAIT show a two to six time improvement over a random baseline at identifying actionable alerts. The ALERTLIFETIME [24] 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 [16] AAIT demonstrated a larger anomaly detection rate curve than the tool ordering of alerts.

Validation of the reported AAITs consisted of an experiment or case study. 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. The current variety of validation techniques preclude a definitive comparison of AAITs reported in literature.

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 18 studies use information about the alerts themselves to predict actionable alerts [4, 16-17, 24-25, 27-28, 30-31, 35]. Six of the selected studies utilize information from the development history of a project to use the past to predict the future [4, 17, 24-25, 35, 43]. Ten of the selected studies use information about the surrounding source code [1, 4-5, 17, 23, 29, 35, 43, 45-46].

10. 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?

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. FAULTBENCH [16] has been proposed to address the need for comparative evaluation of AAITs in the Java programming language. FAULTBENCH provides subject programs and alert oracles for the FINDBUGS [19] static analysis tool. FAULTBENCH could be expanded to evaluate other AAIT by including the history of the software project, as demonstrated in [17], and other ASA like PMD, JLINT, and ESC/JAVA.

11. 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.

12. Appendix: Subject program information

Table 12 presents the subject program information reported in each of the 18 selected studies.

Table 12: 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
Antiword	uk	n/a	C	27	n/a	n/a	ELAN, $EFAN_H$, $EFAN_V$[5]
Apache Web Server	10/29/2003	RETURN VALUE CHECKER	C	200	738	3.69	HISTORYAWARE [43]
Apache XML Security	1.0.4	ESC/JAVA 2	Java	12.4	111	8.95	CHECK 'N' CRASH and DSD-CRASHER [10]
Apache XML Security	1.0.5 D2	ESC/JAVA 2	Java	12.8	104	8.13	CHECK 'N' CRASH and DSD-CRASHER [10]
Apache XML Security	1.0.71	ESC/JAVA 2	Java	10.3	120	11.65	CHECK 'N' CRASH and DSD-CRASHER [10]
Branch 1	uk	SCAS	uk	uk	1692	n/a	SCAS [45]
Branch 2	uk	SCAS	uk	uk	2175	n/a	SCAS [45]
Branch 3	uk	SCAS	uk	uk	3282	n/a	SCAS [45]
CCache	uk	n/a	C	3	n/a	n/a	ELAN, $EFAN_H$, $EFAN_V$ [5]
Check	uk	n/a	C	3	n/a	n/a	ELAN, $EFAN_H$, $EFAN_V$ [5]
Chktex	uk	n/a	C	4	n/a	n/a	ELAN, $EFAN_H$, $EFAN_V$ [5]
Columba	varies	FINDBUGS, PMD, JLINT	Java	uk	uk	n/a	ALERTLIFETIME [24]
Columba	9/11/2003**	FINDBUGS, PMD, JLINT	Java	121	2331****	19.26	HWP [25]
Company X	uk	MC	C	uk	uk	n/a	Z-RANKING [29]
Company X	uk	MC	C	800	640	0.80	FEEDBACK-RANK [29]
Cut	uk	n/a	C	1	n/a	n/a	ELAN, $EFAN_H$, $EFAN_V$ [5]
cvsobjects	0.5beta	FINDBUGS	Java	1.6	7	4.38	APM [16]
Google	uk	FINDBUGS	Java	uk	1652	n/a	RPM08 [35]

Groovy	1.0 beta 1	ESC/JAVA 2	Java	2	34.00	17.00	CHECK 'N' CRASH and DSD-CRASHER [10]
importscrubber	1.4.3	FINDBUGS	Java	1.7	35	20.59	APM [16]
Indent	uk	n/a	C	26	n/a	n/a	ELAN, $EFAN_H$, $EFAN_V$ [5]
iTrust	Fall 2007	FINDBUGS	Java	14.1	110	7.80	APM [16]
jbook	1.4	FINDBUGS	Java	1.3	52	40.00	APM [16]
JBoss JMS	4.0 RC 1	ESC/JAVA 2	Java	5	4	0.80	CHECK 'N' CRASH and DSD-CRASHER [10]
jdom	1.1	FINDBUGS	Java	8.4	55	6.55	APM [16]
jdom	revisions up to 1.1	FINDBUGS	Java	9-13.1***	420****		HW09 [17]
jEdit	varies	FINDBUGS, PMD, JLINT	Java	uk	uk	n/a	ALERTLIFETIME [24]
Lame	uk	n/a	C	27	n/a	n/a	ELAN, $EFAN_H$, $EFAN_V$ [5]
Link	uk	n/a	C	14	n/a	n/a	ELAN, $EFAN_H$, $EFAN_V$ [5]
Linux	2.5.8	MC	C	uk	uk	n/a	Z-RANKING [29]
Linux	2.4.1	MC	C	uk	640	n/a	FEEDBACK-RANK [28]
Lucene	8/30/2004 **	FINDBUGS, PMD, JLINT	Java	37	1513****	40.89	HWP [25]
Memwatch	uk	N/A	C	2	n/a	n/a	ELAN, $EFAN_H$, $EFAN_V$ [5]
Net-tools	1.46	ISA, RATS, ITS4, FLAWFINDER	C	4.1	uk	n/a	ISA [27]
org.eclipse. core.runtime	3.3.1.1	FINDBUGS	Java	2.8	98	35.00	APM [16]
org.eclipse. core.runtime	Revisions up to 3.3.1.1	FINDBUGS	Java	2-15.5***	853****	uk	HW09 [17]
Program A	uk	QA C	C	9	113	12.56	OAY98 [31]
Program B	uk	QA C	C	40	173	4.33	OAY98[31]
Program C	uk	QA C	C	126	1459	11.58	OAY98[31]
Program D	uk	QA C	C	36	1213	33.69	OAY98[31]
Program E	uk	QA C	C	44	1008	22.91	OAY98[31]
Program F	uk	QA C	C	87	2810	32.30	OAY98[31]
Program G	uk	QA C	C	80	779	9.74	OAY98[31]
Pure-ftpd	1.0.17a	ISA, RATS, ITS4, FLAWFINDER	C	25.2	uk	n/a	ISA [27]
rdesktop	uk	BOON	C	17	uk	uk	AJ06 [1]
Scarab	12/10/2002**	FINDBUGS, PMD, JLINT	Java	64	1483****	23.17	HWP [25]
TVoM	uk	QA C	C	91	9740	107.03	BM08B[4]
Uni2Ascii	uk	n/a	C	4	n/a	n/a	ELAN, $EFAN_H$, $EFAN_V$ [5]
Wine	9/14/2004	RETURN VALUE CHECKER	C	uk	2860	n/a	HISTORYAWARE [43]
wu-ftpd	2.5.0	ISA, RATS, ITS4, FLAWFINDER	C	12.4	uk	n/a	ISA [27]
wzdftpd	uk	BOON	C	uk	uk	n/a	AJ06 [1]

13. References

[1] A. Aggarwal and P. Jalote, "Integrating Static and Dynamic Analysis for Detecting Vulnerabilities," *Proceedings of the 30th Annual International Computer Software and Applications Conference*, Chicago, Illinois, USA, September 17-21, 2006, pp. 343-350.

[2] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Looking for Bugs in All the Right Places," *Proceedings of the International Symposium on Software Testing and Analysis*, 2006, pp. 61-71.

[3] C. Boogerd and L. Moonen, "Prioritizing Software Inspection Results using Static Profiling," *Proceedings of the 6th IEEE Workshop on Source Code Analysis and Manipulation*, Philadelphia, PA, USA, September 27-29, 2006, pp. 149-160.

[4] C. Boogerd and L. Moonen, "Assessing the Value of Coding Standards: An Empirical Study," *Proceedings of the IEEE International Conference on Software Maintenance*, Beijing, China, Sept. 28 - Oct. 4, 2008, pp. 277-286.

[5] C. Boogerd and L. Moonen, "On the Use of Data Flow Analysis in Static Profiling," *Proceedings of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, Beijing, China, Sept. 28-29, 2008, pp. 79-88.

[6] B. Chess and G. McGraw, "Static Analysis for Security," in *IEEE Security & Privacy*. vol. 2, no. 6, 2004, pp. 76-79.

[7] B. Chess and J. West, *Secure Programming with Static Analysis*, 1st ed. Upper Saddle River, NJ: Addison-Wesley, 2007.

[8] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Software - Practice and Experience,* vol. 34, no. 11, pp. 1025-1050, 2004.

[9] C. Csallner and Y. Smaragdakis, "Check 'n' Crash: Combining Static Checking and Testing," *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, USA, 2005, pp. 422-431.

[10] C. Csallner, Y. Smaragdakis, and T. Xie, "DSD-Crasher: A Hybrid Analysis Tool for Bug Finding," *ACM Transactions on Software Engineering and Methodology,* vol. 17, no. 2, pp. 1-36, April, 2008.

[11] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions," in *Operating Systems Design and Implementation*no. San Diego, CA, 2000.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Engineering,* vol. 27, no. 2, pp. 99-123, Feb., 2001.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, 2002, pp. 234-245.

[14] E. Haugh and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities," *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, Feb., 2003, pp. 123-130.

[15] S. Heckman and L. Williams, "A Measurement Framework of Alert Characteristics for False Positive Mitigation Models," North Carolina State University TR-2008-23, October 6, 2008.

[16] S. Heckman and L. Williams, "On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques," *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement*, Kaiserslautern, Germany, October 9-10, 2008, pp. 41-50.

[17] S. Heckman and L. Williams, "A Model Building Process for Identifying Actionable Static Analysis Alerts," *Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation*, Denver, CO, USA, 2009, pp. 161-170.

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

[19] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," *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.

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

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

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

[23] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis," *Proceedings of the 12th International Static Analysis Symposium*, Imperial College London, UK, 2005, pp. 203-217.

[24] S. Kim and M. D. Ernst, "Prioritizing Warning Categories by Analyzing Software History," *Proceedings of the International Workshop on Mining Software Repositories*, Minneapolis, MN, USA, May 19-20, 2007, p. 27.

[25] S. Kim and M. D. Ernst, "Which Warnings Should I Fix First?," *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.

[26] 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.

[27] D. Kong, Q. Zheng, C. Chen, J. Shuai, and M. Zhu, "ISA: A Source Code Static Vulnerability Detection System Based on Data Fusion," *Proceedings of the 2nd International Conference on Scalable Information Systems*, Suzhou, China, June 6-8, 2007, p. 55.

[28] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation Exploitation in Error Ranking," *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Newport Beach, CA, USA, 2004, pp. 83-93.

[29] T. Kremenek and D. Engler, "Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations," *Proceedings of the 10th International Static Analysis Symposium*, San Diego, California, 2003, pp. 295-315.

[30] N. Meng, Q. Wang, Q. Wu, and H. Mei, "An Approach to Merge Results of Multiple Static Analysis Tools (short paper)," *Proceedings of the Eight International Conference on Quality Software*, Oxford, UK, August 12-13, 2008, pp. 169-174.

[31] H. Ogasawara, M. Aizawa, and A. Yamada, "Experiences with Program Static Analysis," *Proceedings of the 1998 Software Metrics Symposium*, Bethesda, MD, USA, November 20-21, 1998, pp. 109-112.

[32] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the Bugs Are," *Proceedings of the International Symposium on Software Testing and Analysis*, 2004, pp. 86-96.

[33] C. Pacheco and M. D. Ernst, "Eclat: Automatic Generation and Classification of Test Inputs," *Proceedings of the 19th European Conference on Object-Oriented Programming*, Glasgow, Scotland, July 27-29, 2005, pp. 504-527.

[34] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing Test Cases For Regression Testing," *IEEE Transactions on Software Engineering,* vol. 27, no. 10, pp. 929-948, October, 2001.

[35] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach," *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, May 10-18, 2008, pp. 341-350.

[36] S. E. Sim, S. Easterbrook, and R. C. Holt, "Using Benchmarking to Advance Research: A Challenge to Software Engineering," *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, USA, May 3-10, 2003, pp. 74-83.

[37] W. F. Tichy, "Should Computer Scientists Experiment More?," in *Computer*. vol. 31, no. 5, 1998, pp. 32-40.

[38] S. B. Vardeman and J. m. Jobe, *Basic Engineering Data Collection and Analysis*, 1st ed. Pacific Grove, CA: Duxbury, 2001.

[39] J. Viega, J. T. Floch, Y. Kohno, and G. McGraw, "ITS4: A Static Vulnerability Scanner for C and C++ Code," *Proceedings of the 16th Annual Conference on Computer Security Applications*, New Orleans, LA, USA, Dec.11-15, 2000, pp. 257-267.

[40] D. Wagner, S. F. Jeffrey, E. A. Brewer, and A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," *Proceedings of the Network and Distributed Systems Security Conference*, San Diego, CA, USA, February 2-4, 2000, 2000, pp. 3-17.

[41] S. Wagner and M. A. Florian Deissenboeck, Johann Wimmer, Markus Schwalb, "An Evaluation of Two Bug Pattern Tools for Java," *Proceedings of the 1st IEEE International Conference on Software Testing, Verification, and Validation*, Lillehammer, Norway, to appear, 2008.

[42] D. W. Wall, "Predicting Program Behavior Using Real or Estimated Profiles," *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 26-28, 1991, pp. 59-70.

[43] C. C. Williams and J. K. Hollingsworth, "Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques," *IEEE Transactions on Software Engineering,* vol. 31, no. 6, pp. 466-480, 2005.

[44] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed. Amsterdam: Morgan Kaufmann, 2005.

[45] S. Xiao and C. Pham, "Performing High Efficiency Source Code Static Analysis with Intelligent Extensions," *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, Busan, Korea, November 30-December 3, 2004, pp. 346-355.

[46] K. Yi, H. Choi, J. Kim, and Y. Kim, "An Empirical Study on Classification Methods for Alarms from a Bug-Finding Static C Analyzer," *Information Processing Letters,* vol. 102, no. 2-3, pp. 118-123, 2007.

[47] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, "On the Value of Static Analysis for Fault Detection in Software," *IEEE Transactions on Software Engineering,* vol. 32, no. 4, pp. 240-253, April, 2006.

[48] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects in Eclipse," *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering*, Minneapolis, MN, USA, May 20, 2007, p. 9.