Is There A "Golden" Feature Set for Static Warning Identification?

— An Experimental Evaluation

Junjie Wang[1,3], Song Wang[4], Qing Wang[1,2,3,*]
[1]Laboratory for Internet Software Technologies, [2]State Key Laboratory of Computer Science,
Institute of Software Chinese Academy of Sciences, Beijing, China
[3]University of Chinese Academy of Sciences, Beijing, China
[4]Electrical and Computer Engineering, University of Waterloo, Canada, *Corresponding author
{wangjunjie,wq}@itechs.iscas.ac.cn,song.wang@uwaterloo.ca

ABSTRACT

Background: The most important challenge regarding the use of static analysis tools (e.g., FindBugs) is that there are a large number of warnings that are not acted on by developers. Many features have been proposed to build classification models for the automatic identification of actionable warnings. Through analyzing these features and related studies, we observe several limitations that make the users lack practical guides to apply these features.

Aims: This work aims at conducting a systematic experimental evaluation of all the public available features, and exploring whether there is a golden feature set for actionable warning identification.

Method: We first conduct a systematic literature review to collect all public available features for warning identification. We employ 12 projects with totally 60 revisions as our subject projects. We then implement a tool to extract the values of all features for each project revision to prepare the experimental data.

Results: Experimental evaluation on 116 collected features demonstrates that there is a common set of features (23 features) which take effect in warning identification for most project revisions. These features can achieve satisfied performance with far less time cost for warning identification.

Conclusions: These commonly-selected features can be treated as the golden feature set for identifying actionable warnings. This finding can serve as a practical guideline for facilitating real-world warning identification.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Static analysis, Actionable warning identification, Experimental evaluation

ESEM '18, October 11–12, 2018, Oulu, Finland

ACM ISBN 978-1-4503-5823-1/18/10…$15.00
https://doi.org/10.1145/3239235.3239523

1 INTRODUCTION

Static Analysis (SA) tools (e.g., FindBugs) are widely used to find bugs in software. These tools mainly leverage heuristic pattern matching approaches to scan source code or binary code of a software project, and can infer a wide variety of bugs, security vulnerabilities, and bad programming practices [2, 4, 10]. The widespread adoption of SA tools for entire codebases by commercial software companies provides solid evidence that SA is economically beneficial to find potential defects [3].

The most important challenge regarding the use of SA tools is that there are a large number of warnings that are not acted on by developers. One major reason is the high false positive rate of the reported warnings [10]. Since the software under analysis is not executed, SA tools must speculate on what the actual program behaviors will be. They often over-estimate possible program behaviors, leading to spurious warnings that do not correspond to true defects. In addition, even if the warnings reveal true defects, they can also be ignored. Reasons include warnings implicating obsolete code, "trivial" defects with no impact on the use, and real defects requiring significant effort to fix with little perceived benefit [3, 10]. The large number of unactionable warnings make it time- and effort-consuming to analyze the outcomes generated by SA tools.

To make SA tools more practical, many researches have been proposed to automatically identify actionable warnings [5–7, 11, 12, 14, 15, 17, 18, 26]. Most of them share a common procedure, i.e., propose a set of features and then build machine learning classifiers using these features to predict which warnings are actionable. Through analyzing these studies, we find the following nontrivial limitations.

- **Isolation:** Different studies employed different set of features to predict actionable warnings, and none of them has conducted a complete comparison to explore which features are more effective [5–7, 11, 12, 14, 15, 17, 18, 26].
- **Confliction:** Different studies, even those employed the same features, often evaluated themselves on different projects, and induced conflicting conclusions about features' effectiveness. For example, experiments in [11] suggested that warnings with short lifetime are more likely to be actionable, while [7] found that there is not a clear binary split between the lifetime of actionable and unactionable warnings.

- **Ambiguity:** Features with the same name but different meanings are often observed in different studies. For instance, "warning lifetime" is measured by the number of revisions between the investigated revision and warning-open revision in [7], while it is defined as the amount of time between these two revisions in [12].
- **Coarseness:** Most studies only evaluated the overall performance of their proposed features, rather than the effectiveness of every single feature [5, 6, 11, 15, 26].

Because of the above limitations of existing studies, users lack practical guides when using these features. They would feel confused and unprepared about which features should be applied to warning identification.

Motivated by the limitations, we aim at conducting a systematic experimental evaluation of all the public available features that have been used in the SA Warning Identification (SAWI), and exploring whether these is a golden feature set for actionable warning identification[1]. We will investigate the following two research questions:

- **RQ1**: Is there a common set of features that takes effect in warning identification for most project revisions?
- **RQ2**: What is the performance (i.e., AUC and time cost) of the common feature set in warning identification?

To answer the two research questions, we first conduct an extensive systematic literature review to explore all public available features for warning identification. We employ 12 projects with totally 60 revisions as our subject projects. We then implement a tool to extract the values of all available features for each project revision to prepare the experimental data.

116 features are collected from 10 related studies. They fall into 8 categories, i.e., file characteristics and history; code characteristics, history, and analysis; warning characteristics, history, and combination respectively.

Experimental evaluation on these features demonstrates that there is a common set of features (23 features) that takes effect in warning identification for most project revisions. Most of them belong to warning combination, code characteristics and warning characteristics categories.

We then evaluate the classification performance of these common features from the view of both AUC and time cost. Results reveal that these features can achieve satisfied performance with far less cost for warning identification.

In this sense, these commonly-selected features can be regarded as the golden feature set for warning identification. This finding can serve as a practical guideline for facilitating real-world warning identification.

This paper makes the following contributions:

- A rigorous and extensive evaluation about the golden feature set for warning identification. **To the best of our knowledge, this is the first work to extensively evaluate features for actionable warning identification.**
- 116 public available features for warning identification from a systematic literature review. This can be regarded as the complete set of features for warning identification and will facilitate future research.
- A golden feature set for actionable warning identification. It can serve as the practical guideline to use the public available features.
- Public-access feature dataset and source code for extracting the features[2]. They can facilitate the replication of our study and serve as a benchmark to evaluate other SAWI or bug detection approaches.

The rest of this paper are organized as follows. Section 2 describes the features under experiment. Section 3 presents the subject projects for experiment. Section 4 describes how we prepare the experimental data. Section 5 shows our experimental design. Section 6 presents the results of our research. Section 7 discloses the threats to validity. Section 8 surveys related work. Finally, we summarize this paper in Section 9.

2 FEATURES UNDER EXPERIMENT

To investigate whether there is a golden feature set for actionable warning identification, we first conduct an extensive systematic literature review to explore all public available features (Section 2.1). Secondly, we analyze and summarize all the collected features based on their meanings (Section 2.2).

2.1 Literature Review to Explore Features

We conduct a Systematic Literature Review (SLR) to collect features which have been used in SAWI. We use the SLR guidelines described by Kitchenham [13] to develop our SLR protocol, which is described in the following subsections.

2.1.1 ***Search Terms.*** We identify key terms used for the search from previous work [8] and our experience with the subject area. The search terms are as follows: *static analysis and (alert or warning or bug or defect or fault) and (identification or classification or detection or prediction or prioritization or ranking or reduction).*

2.1.2 ***Sources of Information.*** The following four databases are covered for retrieving related literatures: *ACM Digital Library, IEEE Xplore, Springer Link*, and *ScienceDirect*. The searches were conducted in May 2016.

2.1.3 ***Study Selection.*** Selection of studies for inclusion in the SLR is a three-stage process: (1) initial selection of studies based on the title; (2) selection of studies after reading the abstract; (3) further selection of studies after reading the paper.

Since our SLR aims at selecting the studies which have proposed features for actionable warning identification, we introduce the following inclusion and exclusion criteria to facilitate the selection process. Studies selected at each stage of the selection process meet our **inclusion criteria**:

- Full and short peer-review papers with empirical results.
- Work on the results of SA tools.
- Focus on the automatic identification about whether a single SA warning or a group of warnings are actionable or unactionable.

[1]Note that, this paper experiments with FindBugs (see Section 5 for details), which is the most commonly-used SA tool, so in the following paper, the *SA warning* means the warning generated by FindBugs.

[2]https://github.com/wangjunjieISCAS/SAWarningIdentification

- Propose at least one feature (or factor, characteristic, attribute) to conduct the identification.
- Focus on general types of warnings in Java language.

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

- Papers unrelated to static analysis or actionable warning identification.
- Refinements or enhancements to SAWI approaches.
- Theoretical papers about SAWI (i.e., no empirical result).
- Studies focusing on specific types of warnings (e.g., security vulnerability) or specific types of programming languages (except Java language).
- Dynamic analysis.

Our initial protocol of inclusion and exclusion criteria are borrowed from [8]. During the reviewing process, new criteria are added. For example, we noticed several papers focusing on specific types of warnings (e.g., [22] studied security vulnerability only). Through reviewing them, we found that the proposed features are not appropriate for identifying common types of warnings. Besides, since this work focuses on evaluating the features' effect on Java projects. We also exclude the studies particularly focusing on other types of languages (e.g., [25] focused on C language).

In summary, at stage 1, we started with *18,492* distinct papers from the database search and *624* of them were selected and moved to stage 2 for further selection. These studies are reviewed by their abstracts and *58* papers with relevant abstracts are selected for further reading in stage 3. *10* papers were finally selected.

2.1.4 ***Selection Verification***. The first author did the selection of the studies following the process outlined before. The second author provided validation of the studies at each stage of the selection process. After stage 1, 1,000 (5%) of the studies were randomly selected for the second author to validate. The first author prepared the selection, ensuring that the sample had approximately the same proportion of selected and rejected studies as the full population. 968 (96%) of the studies had the same selection by the first and second author. All the differences were between the studies the first author selected but the second author did not. The discrepancy is caused by the differences between the interpretations of the refinements to SAWI approaches. However, we found that all these papers were not included in the final set, so it did not influence the results.

After stage 2, the second author evaluated the abstracts for 50 (8%) randomly selected studies. We again ensured that the sample had approximately same distribution of selected and rejected studies as the full population of studies. 46 (92%) of the randomly selected studies have the same classification by the two authors. Similarly with stage 1, most of the differences came from these studies the first author classified as "1" but the second author classified as "0". One study, which was classified as "1" by the second author and "0" by the first author, was included in stage 3 of study selection. It is worth mentioning that this paper was not included in the final set.

We also conducted similar selection verification process for stage 3, and all the randomly selected studies (10, 17%) has the same selection by the two authors.

2.1.5 ***Study Quality Assessment***. For each of the final selected studies, we answered the questions below to assess its quality.

- Is there a clearly stated research goal related to SAWI?
- Is there a defined and repeatable SAWI technique?
- Are there plenty of features proposed for SAWI?
- Are the limitations to the SAWI enumerated?
- Is there a clear methodology for validating the SAWI?
- Are the subject programs selected for validation suitable (e.g., large enough to demonstrate the efficacy of the technique) for the research goals?
- Are there control techniques or baselines to demonstrate the effectiveness of the SAWI?
- Are the evaluation metrics relevant (e.g., evaluate the effectiveness of the SAWI) to the research objectives?
- Are the presented results clear and relevant to the research objectives stated in the study?
- Is there any explicit contribution to SAWI?

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. Five papers received a quality score of 10, and the average quality score was 8.6. This indicates the high quality of the selected studies.

2.1.6 ***Data Extraction***. For each of the selected studies, we extracted the following data: *feature name, category, meaning, method to extract it.*

The features are then synthesized. For features with the same name and the same meaning from different studies, we just merge them into one feature. For two features with the same name but different meanings, we rename them and keep them as two different features, e.g., F34 and F40 in Table 1. For two features with different names but the same meaning, we rename one of the features and merge them as one feature, e.g., F21 in [7] and [17].

2.2 Overview of Collected Features

After the data extraction in Section 2.1.6, 116 features (summarized in Table 1) are collected from the 10 papers selected in the SLR. The features fall into 8 categories related with file, code and warning. For the feature that has specified category in its original study, we reuse the category. Otherwise, we manually group it into one of the below categories.

File characteristic (fChr) and file history (fHst): **fChr** category contains features about the static characteristics of the file where the warning locates, e.g., *F1–file name*. **fHst** category contains features about the change history of a warning file, e.g., *F9–the latest modification revision of a file.*

Code characteristic (cChr), code history (cHst), and code analysis (cAnl): **cChr** category contains features about the static characteristics of the source code of the file where a warning locates, e.g., *F22–number of comment lines of the code*. **cHst** category contains features about the modifications in lines of code in a warning file, e.g., *F34–number of added lines in file during the past three months*. **cAnl** category contains features about the program analysis patterns mined from the source code in a warning file, e.g., *F70–the name of the method being called.*

Table 1: Overview of the collected features

Id	Name; *Meaning*	Cat.	Reference	#Rev.
F1, F2, F3*	file name, package name, project name; *the file name, package name, project name where the warning locates*	fChr	[6, 7, 14, 26]	17,15
F4	file type; *file extension name*	fChr	[7, 17]	1
F5*	SA tool name	fChr	[15]	n/a
(F6-F8)*	location, file, project warnings for tool; *warnings reported for the same location, file, project by each tool*	fChr	[15, 26]	n/a
F9, F10, F11*	latest file, package, project modification; *latest modification revision*	fHst	[7]	15,9
F12, F13, F14*	file, package, project staleness; *amount of time between current revision and last modification revision of file, package, project*	fHst	[7, 17]	12,11
F15	file age; *number of days the file has existed*	fHst	[17]	18
F16, F17	file creation, deletion revision	fHst	[7, 11]	24,8
F18	developers; *set of developers who have made changes to the file*	fHst	[7]	26
F19, F20, F21	method, file, package size; *number of non-comment source code statements in method, file, package*	cChr	[7, 17]	11,16,15
F22	comment length; *number of comment lines in file*	cChr	[15]	11
F23	comment-code ratio; *ratio of comment length and code length in file*	cChr	[15]	18
F24, F25	method, file depth; *depth of warned line in method, file*	cChr	[15]	24,20
F26, F27	method callers, callees; *number of callers, callees of warned method*	cChr	[15]	9,17
F28, F29	methods in file, package; *number of methods in file, package*	cChr	[7]	18,12
F30, F31	classes in file, package; *number of (inner) classes in file, package*	cChr	[7]	16,22
F32	indentation; *spaces indenting warned line*	cChr	[17]	16
F33	complexity; *cyclomatic complexity*	cChr	[7]	8
F34-F39	added, changed, deleted, growth, total, percentage of lines of code in file during the past 3 months	cHst	[17]	10-17
F40-F45	added, changed, deleted, growth, total, percentage of lines of code in file during the past 25 revisions	cHst	[7]	9-22
F46-F51	added, changed, deleted, growth, total, percentage of lines of code in package during the past 3 months	cHst	[17]	12-18
F52-F57	added, changed, deleted, growth, total, percentage of lines of code in package during the past 25 revisions	cHst	[7]	10-17
(F58-F63)*	added, changed, deleted, growth, total, percentage of lines of code in project during the past 3 months	cHst	[17]	n/a
(F64-F69)*	added, changed, deleted, growth, total, percentage of lines of code in project during the past 25 revisions	cHst	[7]	n/a
F70-F73	call name, class, parameter signature, return type; *name of method being called, name of class containing the method*	cAnl	[5]	14,11,24,15
F74, F75	new type, new concrete type; *class, or concrete type of object being created*	cAnl	[5]	9,9
F76	operator; *operator for the binary operation*	cAnl	[5]	14
F77, F78	field access class, field; *class containing the field being accessed, name of field being accessed*	cAnl	[5]	3,1
F79	catch; *whether a catch statement is present*	cAnl	[5]	12
F80-F83	field name, type, visibility, is static/final	cAnl	[5]	0,4,2,1
F84-F86	method visibility, return type, is static/final/abstract/protected	cAnl	[5]	19,16,0
F87, F88	class visibility, is abstract/interface/array class	cAnl	[5]	8,0
F89-F92, F93*	warning pattern, type, priority, rank, range (Google warning descriptors)	wChr	[6, 7, 11, 17]	18,18,24,13
F94-F96	warnings in method, file, package; *number of warnings in method, file, package*	wChr	[6, 7, 17]	0,17,18
F97	warning modifications; *number of times the warning's line number has changed*	wHst	[7]	11
F98	warning open revision	wHst	[7, 11]	15
F99, F100	warning lifetime by revision, by time; *number of revisions, amount of time between current revision and open revision*	wHst	[7, 11, 26]	25,14
F101*	developer idea; *four different idea: fix, not a problem, ignore, analyse*	wHst	[26]	n/a
F102	size context for a warning type; *number of warnings from a warning type normalized by S; S is total number of warnings*	wCmb	[6, 7]	16
F103-F105	size context in method, file, package; *number of warnings in the method, file, package normalized by S*	wCmb	[6, 7]	14,11,15
F106	warning context for warning type; *difference of actionable and unactionable warnings for a warning type normalized by S*	wCmb	[6, 7]	20
F107-F109	warning context in method, file, package; *difference of actionable and unactionable warnings for the method, file, package normalized by S*	wCmb	[6, 7]	30,26,10
F110, F111	fix, non-fix change removal rate; *warnings in a type that is removed by bug-fix commit, non-bug-fix commit normalized by S*	wCmb	[12]	14,12
F112	defect likelihood for warning pattern; $D(P_{ij}) = T_{ij}/(T_{ij}+F_{ij})$, *where* P_{ij} *denotes the* j_{th} *warning pattern in* C_i *warning type,* T_{ij} *is number of actionable warnings for pattern* P_{ij}, F_{ij} *is number of unactionable warnings for pattern* P_{ij}	wCmb	[18]	23
F113	variance of likelihood; *variance of* $D(P_{ij})$ *for each warning pattern* P_{ij} *in a warning type* C_i	wCmb	[18]	17
F114	defect likelihood for warning type; $D(C_i) = D(P_{i1})*S_{i1} + \cdots + D(P_{in})*S_{in}$, *where* $S_{ij} = T_{ij}+F_{ij}$	wCmb	[18]	13
F115	discretization of defect likelihood; *discretization of* $D(C_i)$ *for each warning type* C_i *in the project*	wCmb	[18]	23
F116	average lifetime for warning type; *average value for feature F100 for a warning type*	wCmb	[11]	22

Note: The features marked with * denote they will not be experimentally investigated in this paper.
#Rev. is the number of project revisions in which the features are selected (details are in Section 6.1).

Warning characteristics (wChr), warning history (wHst), and warning combination (wCmb): **wChr** category contains features about the characteristics of a warning, e.g., *F89–warning pattern*. **wHst** category contains features about the change history of a warning itself, e.g., *F98–warning open revision*. **wCmb** category contains features concerning the combination of warning characteristics and other kinds of information, e.g., *F106–warning context for warning type*. This feature reflects the percentages of actionable warnings and unactionable warnings of a specific type (e.g., Correctness) during the evolution of a project.

Note that, not all collected features are applicable for this work, which are marked with an asterisk (*) in Table I. Some of them are about the status of the whole project (e.g., *F3–project name*), thus all warnings of a project possess the same feature value. Hence these features are useless for our within-project evaluation setting. Others can only be collected in a specific context (e.g., *F93 is a warning descriptor in Google*). Thus, we will not experimentally investigate these features, and only use the remaining 95 features (i.e., features without * in Table 1) for experiments. In the following paper, "all features" or "all collected features" refer to these 95 features.

3 SUBJECT PROJECTS

Our experiment is conducted on 12 subject projects listed in Table 2. These projects are selected because of their sizes (they are large enough to have many SA warnings), ages (they have source code histories spanning multiple years) and the fact that they have Git (a version control system) repositories which makes it easier to extract features. The data are collected in September and October of 2016. To ensure the selected project revisions are different enough with each other so as to make solid conclusions, we set different revision intervals for different projects, e.g., 3 months for Lucene-solr and 6 months for Maven.

We randomly separate the projects into two datasets, and each dataset has 6 projects (as showed in Table 2). We use dataset1 to answer RQ1, while use dataset1 and dataset2 to answer RQ2 to avoid overfitting.

Note that, we have tried several other projects (e.g., jdom, aspectJ, openEJB, etc.), which were used in the studies where the features come from. They are not included in this paper because they have few number of warnings (i.e., less than 100). We assume there is little value for these projects to conduct the warning identification.

Table 2: Subject projects

Data	Project	Domain	Total Revisions	Start Revision	End Revision	Revision Interval	#Rev	Size (KLOC)	Warnings (min-max)	Act. Warn (min-max)	Unact. Warn (min-max)	Del. Warn (min-max)	#Sel. Feature (min,avg,max)
dataset1	Lucene-solr	Search engine	26,840	2013.01	2014.01	3 month	5	283-329	3553-3865	1202-1235	1843-2257	440-475	36, 42, 52
dataset1	Tomcat	Server	18,068	2013.01	2014.01	3 month	5	179-184	1435-1500	326-492	978-1054	0-0	15, 38, 58
dataset1	Maven	Project manage	10,218	2012.01	2014.01	6 month	5	54-55	809-889	28-111	651-790	44-67	15, 24, 36
dataset1	Poi	File formatting	8,793	2012.01	2014.01	6 month	5	410-443	2547-2759	498-576	2042-2177	6-7	50,63,74
dataset1	Derby	Database	8,135	2013.01	2014.01	3 month	5	219-238	2457-2603	121-450	2149-2386	0-4	11, 28, 48
dataset1	Phoenix	Driver	2,077	2013.01	2014.01	3 month	5	99-172	1147-2402	316-422	819-2046	11-13	45,49,55
dataset2	Cassandra	Big data manage	22,580	2013.01	2014.01	3 month	5	347-362	2298-2665	356-830	1388-2245	54-87	38, 54, 77
dataset2	Jmeter	Performance manage	14,127	2012.01	2014.01	6 month	5	71-75	598-652	145-206	376-468	7-16	17, 27, 36
dataset2	Ant	Build manage	13,581	2012.01	2014.01	6 month	5	92-95	1115-1229	54-378	815-1061	0-4	21, 32, 44
dataset2	Log4j2	Logging Service	9,379	2012.01	2014.01	6 month	5	24-116	485-1063	175-366	226-652	32-114	9,25,34
dataset2	Qpid	Messaging	7,831	2013.01	2014.01	3 month	5	196-200	1812-2052	275-400	1376-1586	6-129	14,27,51
dataset2	Commons .lang	Java utility	4,862	2012.01	2014.01	6 month	5	49-55	534-786	42-183	349-744	0-2	4, 16, 47

4 DATA PREPARATION

To investigate whether there is a golden feature set, we need to prepare the experimental data. In detail, we need to extract the values of all available features for each project revision. Since most of the existing studies did not release their tools for feature extraction, we implement our own tool to extract the values for each feature, strictly following the studies where the features come from.

We prepare data by using the the following steps.

- For a project under investigation, use *git log* to retrieve all commit logs from its Git repository.
- For a specific revision of the project under investigation, use *git checkout* to obtain the source files.
- Compile the source code.
- Run FindBugs on the compiled code and generate a list of warnings for the revision.
- Retrieve the commit types (bug-fixing or non-bug-fixing, which are used to extract features F110, F111) using the issue types recorded in issue tracking system (Jira or Bugzilla). This is done based on the link established by issue id both in the commit message and issue tracking system, which is a common practice [6, 11, 12].
- Extract all the original data fields showed in Table 3 from the warning information generated by the SA tool, the commit logs and source code files.
- Obtain the value of features based on the data fields, with details shown below.

Table 3: Data source and data field for feature evaluation

Data	Source	Data field
Warning	SA tool	warning pattern, type, priority, rank, method name, file name, file type, warned line number
Commit log	Version control system	changed file, change time, revision number, added lines of code, deleted lines of code
Source code	Version control system	package name, file name, method name, code, line number

Note that, to differentiate the *source code* in *data* column and *source code* in *data field* column, we only use *code* to refer the data field.

File characteristics (fChr) and warning characteristics (wChr): these features can be extracted directly from the data fields in *warning* (e.g., F89–warning pattern) and data fields in *source code* (e.g., F2–package name).

Code characteristics (cChr): these features can be extracted based on the data fields in *source code* using JavaNCSS[3] tool, as introduced in [7, 15]. For example, taking the *code* of a file as input, JavaNCSS can obtain the comment length (F22) automatically.

File history (fHst): these features can be extracted based on the data fields of *changed file*, *change time*, and *revision number in commit log*. Take F9 (latest modification revision of a file) as an example, we just retrieve the *revision number* when the particular file (*changed file*) was latest modified (*change time*) from all the commit logs of the project.

Code history (cHst): these features can be extracted based on the data fields in *commit log*. Based on the data fields of *added lines of code, deleted lines of code*, all other types of code changes can be inferred. For example, to extract F34 (number of added lines in file during the past three months), we first retrieve all the commit logs in which the particular file (*changed file*) was modified during the past three months (*change time*). Then we sum *the added lines* in these commits.

Warning history (wHst): these features can be extracted based on the data fields of *warned line number*, data fields in *commit log* and *source code*. Take F89 (warning open revision) as an example, we first identify the exact code by matching *warned line number* to *code*, then we use the data fields in *commit log* to locate the *revision number* where the exact code is initially added (*added lines of code*).

Code analysis (cAnl): these features can be extracted from the data fields of *code* and *warned line number*. We first retrieve a set of related code statements, which is potentially relevant to the warned code. We use the T. J. Watson Libraries for analysis (WALA)[4] and Eclipse JDT library[5] for Abstract Syntax Tree (AST) generation, as introduced in [5]. Then we traverse ASTs and use the warned code snippets as seeds for program slice construction. The features are finally extracted based on the related code statements, as well as the class hierarchy of the subject program (details are in [5]).

Warning combination (wCmb): these features can be extracted from the data fields in *warning*, *commit log*, and *source code*. For features of F106-F109 and F112-F115, we first need to label each warning as actionable or unactionable. Detailed methods are shown in Section 5.2. For features F110 and F111, we need to label each commit as a bug-fixing commit or a non-bug-fixing commit, which is obtained from the issue tracking system.

[3]http://www.kclee.de/clemens/java/javancss/
[4]http://wala.sourceforge.net/wiki/index.php/
[5]https://www.eclipse.org/jdt/

5 EXPERIMENT DESIGN

5.1 Static Analysis Tools

Typical static analysis tools include FindBugs, PMD, Jlint, etc. Previous work [16] has shown that FindBugs is more effective than PMD and Jlint in terms of detection precision. Thus, in our experiments, we use FindBugs[6] as the subject SA tool. Future work will explore other SA tools.

FindBugs is an open-source static analysis tool for Java programs. The tool analyzes Java bytecode with 424 bug patterns. These patterns are organized into nine types: Bad Practice (questionable coding practices), Correctness (suspected defects), Experimental, Internationalization, Malicious Code Vulnerability, Multithreaded Correctness, Performance, Security, and Dodgy Code (confusing or anomalous code). We use all these reported warnings to evaluate the collected features.

5.2 Ground Truth Building

We also need to accurately label warnings as actionable or unactionable to build the ground truth. In this work, we use a commonly-used method [5, 6, 15] to build the ground truth. The general idea is that if a warning disappears in a later revision, it is treated as actionable; otherwise, it is unactionable. The process is as follows:

- Choose a number of revisions across a project's history starting from the *start revision* (shown in Table 2) to *two years after the end revision* (to ensure the trustworthy of labeling results) with a specific *revision interval* (in Table 2).
- Run FindBugs on the compiled code of each revision and generate a list of warnings for each revision (as Section 4 describes).
- For a specific warning, find whether it is closed in later revisions, i.e., it is no longer present in later revisions.
  - If a warning is closed, label it as actionable;
  - If a warning is still present until the last revision of our collected data, label it as unactionable;
  - If the file containing a warning is deleted, label it as unknown and remove it from the list.

Note that, there are at least two years between commit time and data collection time, so the labelling results are trustworthy. Detailed statistics of warnings for each project are shown in Table 2 column 9-12.

5.3 Experiment Method for RQ1

To answer RQ1, we first conduct the feature selection to select features which take effect in each project revision. We employ the greedy backward elimination algorithm for feature selection [23], which is a common practice in feature selection of software engineering tasks [24]. The algorithm begins with all the investigated features. At each iteration, it builds a machine learning classifier to conduct the warning classification, and greedily removes a feature from the current set of features such that the performance of classification on the dataset is maximized. The feature set that obtains the best performance across all iterations is returned. During the feature selection process, we employ Random Forest as the machine learning classifier and AUC as the evaluation metric for classification performance (details are in Section 5.4).

Secondly, based on the selected features for each revision, we find out the commonly-selected features, which are selected in most circumstances (more than 60% experimental revisions).

5.4 Experiment Method for RQ2

To answer RQ2, we build classification models by using the commonly-selected subset of features and record the performance of these features for actionable warning identification.

We also employ other four treatments to serve as the baselines. Details of the five treatments are shown below:

Commonly-selected features (*common* for short): Use the subset of features that are common for most revisions (features in Table 4).

Total features (*total* for short): Use all features shown in Table 1, which is a common practice in previous studies [5, 15].

Prior-specific features (*prior* for short): Use the subset of features selected based on the prior revision of the specific project, which is another commonly-used validation method in previous researches [17, 24].

Current-specific features (*current* for short): Use the subset of features selected based on the experimental revision and conduct 10-fold cross validation within the revision. Note that this scenario is the theoretical best performance, yet is actually unattainable in real practice.

Random features (*random* for short): Randomly choose 23 features (equal number with commonly-selected features) from total features and conduct warning classification. Repeat the experiment for 10 times and obtain the average performance.

We have mentioned that *current* employs the cross validation within the revision. Other four treatments all involve cross-revision validation within a project, which is a most common evaluation scenario [5, 6, 15]. In detail, we build a classifier in a prior revision, and use a later revision to measure the performance. Since the first revision do not have a prior revision, for each project, we conduct four successive cross-revision experiments.

All these experiments involve utilizing machine learning classification algorithms to build classifiers. We experiment with six frequently-used machine learning algorithms [23], i.e., Random Forest, Decision Tree (*J48* for short), Boost, Naive Bayes, Logistic Regression (*LR* for short), Support Vector Machine (*SVM* for short).

For the classification performance, we use the *AUC*, which is a widely adopted metric especially for imbalance data [5, 7, 15, 26]. It is the area under ROC curve, which measures the overall discrimination ability of a classifier [23]. To investigate the cost of warning identification, we record the feature extraction time and feature selection time of each treatment.

6 RESULTS AND ANALYSIS

6.1 Answering RQ1

To answer this question, we first conduct the feature selection process for each project revision as introduced in Section 5.3. Table 2 (the last column) presents a brief overview of the number of selected features for each project. Table 1 (the last column) demonstrates the number of project revisions in which a feature is selected.

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

Table 4: Commonly-selected features (RQ1)

Cat.	Feature	#Rev. (Revision Information)
wCmb ***6/15=40%***	F107: warning context in method	30(*lu:5, tm:5, mv:5, dr:5, po:5, ph:5*)
	F108: warning context in file	26(*lu:2, tm:5, mv:5, dr:4, po:5, ph:5*)
	F112: defect likelihood for warning pattern	23(*lu:3, tm:5, mv:2, dr:3, po:5, ph:5*)
	F115: discretization of defect likelihood	23(*lu:5, tm:5, mv:1, dr:4, po:5, ph:3*)
	F116: average lifetime for warning type	22(*lu:5, tm:3, mv:1, dr:3, po:5, ph:5*)
	F106: warning context for warning type	20(*lu:5, tm:4, mv:2, dr:0, po:5, ph:4*)
cChr ***5/15=33%***	F24: method depth	24(*lu:5, tm:5, mv:1, dr:4, po:4, ph:5*)
	F31: classes in package	22(*lu:4, tm:5, mv:3, dr:2, po:5, ph:3*)
	F25: file depth	20(*lu:5, tm:2, mv:3, dr:2, po:5, ph:3*)
	F23: comment-code ratio	18(*lu:0, tm:3, mv:3, dr:5, po:4, ph:3*)
	F28: methods in file	18(*lu:0, tm:4, mv:4, dr:1, po:4, ph:5*)
wChr ***4/7=57%***	F91: warning priority	24(*lu:5, tm:4, mv:4, dr:3, po:5, ph:3*)
	F89: warning pattern	18(*lu:4, tm:4, mv:1, dr:2, po:5, ph:2*)
	F90: warning type	18(*lu:0, tm:5, mv:3, dr:1, po:5, ph:4*)
	F96: warnings in package	18(*lu:1, tm:5, mv:0, dr:2, po:5, ph:5*)
fHst ***3/8=37%***	F18: developers	26(*lu:5, tm:5, mv:1, dr:5, po:5, ph:5*)
	F16: file creation revision	24(*lu:5, tm:5, mv:2, dr:4, po:5, ph:3*)
	F15: file age	18(*lu:1, tm:3, mv:2, dr:3, po:5, ph:4*)
cAnl ***2/19=10%***	F72: parameter signature	24(*lu:5, tm:5, mv:2, dr:2, po:5, ph:5*)
	F84: method visibility	19(*lu:5, tm:5, mv:0, dr:2, po:4, ph:3*)
cHst ***2/24=8%***	F40: added lines of code in file during the past 25 revisions	22(*lu:5, tm:5, mv:1, dr:3, po:5, ph:3*)
	F46: added lines of code in package during the past 3 months	18(*lu:1, tm:4, mv:2, dr:2, po:4, ph:5*)
wHst ***1/4=25%***	F99: warning lifetime by rev	25(*lu:5, tm:4, mv:4, dr:3, po:5, ph:4*)
fChr(0)		

P:X denotes the feature is selected in X revisions of project P.
Projects are abbreviated as follows: **lu**:Lucene-solr, **tm**:Tomcat, **mv**:Maven, **dr**:Derby, **po**:Poi, **ph**:Phoenix.

Based on the selected features for each project revision, we further find out the features that take effect in most experimental revisions (more than 60% revisions). Table 4 demonstrates the commonly-selected features with feature name, category, the percentage of selected features for each category, how many and which revisions the feature is selected. In total, 23 features involving 7 categories are selected.

We can easily observe that *warning combination*, which contributes six features (40% of its total features), is the most important category. This implies that the features combined of warning characteristics and other information can act as the crucial indicators for warning identification. This is why several researches only use the features in this category for warning identification and can also achieve relatively good performance [6, 11]. We notice that these features rely on the software process information (e.g., whether certain warnings are proved to be actionable during project history), which has been proven to be crucial for defect prediction [16]. Besides, *warning context in method* is selected by all experimental project revisions, and *warning context in file* is selected by 90% revisions. This suggests that if there were large percentage of actionable warnings in the method (or file) where a new warning locates, then the warning will have a higher probability to be actionable.

The second important category is *code characteristics*, which contributes five features (33% of its total features). We infer that the static characteristics of source code also act as the essential indicators for warning identification. Some of the commonly-selected features coincide with the static features for defect prediction [20], which have been proven to be effective in indicating software bugs.

The third important category is *warning characteristics*. Although it only contributes four features, these features account for 57% of its total features, which suggests the uniqueness of this category. It is reasonable because the characteristics of warnings have the most direct relationship with actionable warnings. The selected features include *warning priority, warning pattern, warning type, warnings in package*. We can infer that certain *warning type* or *warning pattern* might be more likely to be actionable than others.

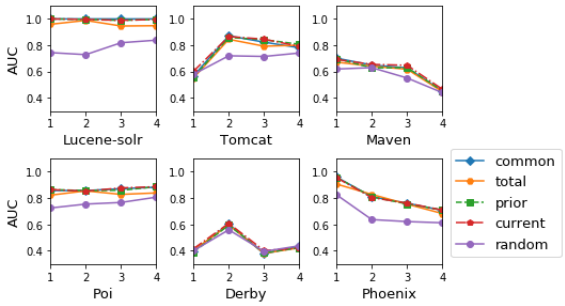

Figure 1: Performance of features on dataset1 (RQ2)

File history is another important category. 37% of its total features are selected, including *developers, file creation revision,* and *file age*. Features in this category measure the change history of the files where the warning code locates, which further indicates the importance of software process information.

Furthermore, *code analysis* and *code history* are also necessary categories for warning classification. Features in *code analysis*, e.g., *parameter signature* and *method visibility*, model the program analysis patterns. Although the warnings are generated based on the static analysis information, existing features in this category cannot effectively capture the property of actionable warnings as only 2 features (10% of total features) are selected. Features in *code history* describe the changes in lines of code if the file or package where a warning code locates. There are 24 features proposed in previous studies, while only 2 features are commonly-selected. We can infer that we do not need to extract all these features for actionable warning identification. Furthermore, the selected 2 features are about the added lines of code. It might because when a file or package is enhanced, the related warnings would be more likely to be fixed in the meantime.

> There is a common set of features that take effect in warning identification on most project revisions. Most of them belong to warning combination, code characteristics, and warning characteristics categories.

6.2 Answering RQ2

To answer RQ2, we not only provide the classification performance of different treatments (Section 6.2.1) and the performance under different classifiers (Section 6.2.2), but also conduct the time cost evaluation (Section 6.2.3) to further evaluate the effectiveness of commonly-selected features.

6.2.1 ***Performance Comparison of Different Treatments.*** Figure 1 and 2 present the AUC under the Random Forest classifier for each of the five treatments introduced in Section 5.4. As the commonly-selected features are selected based on dataset1, we conduct the warning classification on both dataset1 and dataset2 to avoid overfitting.

We can easily observe that for both dataset1 and dataset2, the random features achieve the lowest performance. This indicates the necessity to conduct feature selection. Furthermore, we also find that the classification performance based on total features

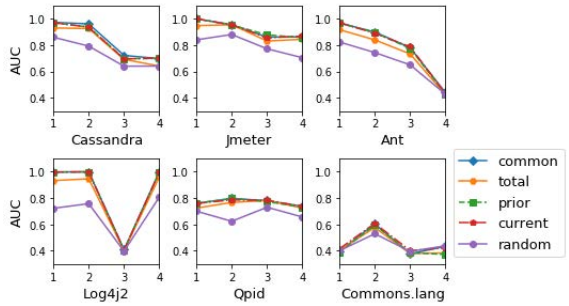

Figure 2: Performance of features on dataset2 (RQ2)

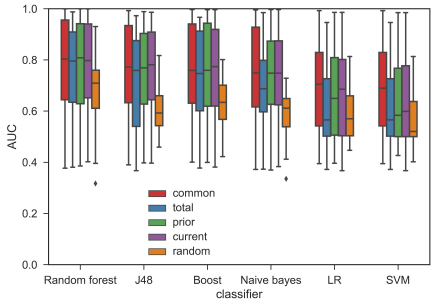

Figure 3: Performance of different classifiers (RQ2)

is slightly worse than the performance using commonly-selected features. This maybe because redundant features may bring noise to the classification, and further indicate the need for refined feature set.

Commonly-selected features, current-specific features, and prior-specific features all involve careful and rigorous feature selection process. We do not observe the difference in their classification performance from Figure 1 and 2. When using the commonly-selected features for warning identification, the median AUC for dataset1 is 0.79 while the median AUC for dataset2 is 0.82. We notice that for some projects (e.g., Log4j2), the AUC among different revisions are not stable. We manually checked the commits logs around the time of the 3rd experimental revision of Log4j2 (which has a low AUC), and found that the project undergone enormous changes during that time. Large number of files were deleted, while many new files were added. This might be the root cause of the performance decline of Log4j2. We also notice this happens for other treatments. This indicates that the low classification performance can not reflect the ineffectiveness of commonly-selected features.

We further conduct Mann-Whitney U test among the AUC for the five treatments. Results turn out that, except random features, the p-values between other pair of treatments are all higher than 0.05, suggesting there is no significant difference among these treatments. This further indicates the effectiveness of the commonly-selected features for warning identification.

6.2.2 ***Performance Comparison of Different Classifiers.*** In figure 3 we present the classification performance for different machine learning classifiers. We can observe that the best performance is obtained by the Random Forest classifier, following by J48, Boost and Naive Bayes. This is reasonable because Random Forest is an ensemble learning method which combines the predictive effect of numerous single classifier.

We then compare the performance of different treatments under these six classifiers. No matter which machine learning classifier is used, the classification based on commonly-selected features is superior to, or equal with other experiment treatments.

Table 5: Time cost in hour (RQ2)

Project	Setting	ET	ST	Project	ET	ST
common	Lucene-solr	1.1	0	Cassandra	0.9	0
total		4.2	0		2.7	0
prior		4.2	26		2.7	14
common	Jmeter	0.6	0	Tomcat	0.8	0
total		1.8	0		2.9	0
prior		1.8	9		2.9	13
common	Ant	0.8	0	Maven	0.8	0
total		2.5	0		2.1	0
prior		2.5	12		2.1	11
common	Poi	0.9	0	Log4j2	0.8	0
total		3.7	0		2.0	0
prior		3.7	24		2.0	10
common	Commons.lang	0.7	0	Derby	0.9	0
total		1.9	0		2.8	0
prior		1.9	9		2.8	15
common	Phoenix	0.8	0	Qpid	0.8	0
total		2.8	0		2.3	0
prior		2.8	14		2.3	12

6.2.3 ***Time Cost Evaluation.*** Since there is no significant difference among these experiment treatments, we additionally perform the time cost evaluation. It is performed on a computer with CPU Intel(R) Core(TM) i7 2.5 GHz PC with 8GB RAM running Windows7 OS (64-bit). For each treatment and each project, we record the average feature extraction time and feature selection time for all revisions of the project.

From Table 5, we can easily observe that when using all features (*total features*), we need several hours to extract the values of these features. When using features selected based on prior project revision (*prior-specific features*), we not only need several hours to extract features, but also need additional several tens of hours to conduct feature selection. However, when using these *commonly-selected features*, we can save the time both for feature selection and for feature extraction. In detail, we do not need to conduct the feature selection as *prior-specific features*. For feature extraction, it only requires about 30% of the time compared with *total features* and *prior-specific features*.

Note that, due to space limit, we do not present the time cost for the treatment of *current-specific features* (only employed for the theoretical best performance) and *random features* (the performance is the worst).

Furthermore, besides the explicit time for extracting and selecting features, the users also need extra time to learn how to conduct feature extraction and selection. When using the commonly-selected features, the time spent in learning can also be reduced.

Using the commonly-selected features for warning identification can achieve satisfied performance with far less cost. In this sense, these commonly-selected features can be regarded as the golden feature set for warning identification.

7 THREATS TO VALIDITY

Construct validity of this study mainly questions the selection of studies. It is addressed through specifying a research protocol that defines the search terms, selection strategy, inclusion and exclusion criteria, as well as quality assessment. We also employ the selection verification process to let the second author review some sampled studies to further minimize the risk of exclusion of relevant studies. In addition, our evaluation is only conducted in terms of one SA tool (i.e., Findbugs), and the golden feature set might not perform well for other SA tools. Besides, although we employ a commonly-used feature selection methods, other feature selection methods might come to a slight different golden feature set. Further exploration is needed for other SA tools (e.g., PMD, Jlint) and other feature selection methods.

The internal threats concern the implementation of feature extraction. We strictly follow the procedures described in the original studies where the features come from, and test the implementation based on 326 test cases to ensure the correctness of feature extraction. Furthermore, for features in code analysis category, we re-use the code[7] provided by the author. In addition, we utilize the link established by issue id in the commit message and issue tracking system, which is a common practice [6, 11, 12], in feature extraction. However, there could be impression considering various cases as the typographical error in issue id, duplicate commits, etc.

The external threats concern the generality of this study. Our dataset consists of 12 projects from various domains and in relatively large size, which can help reduce this threats.

8 RELATED WORK

Section 2 has introduced many actionable warning identification approaches [5–7, 11, 12, 14, 15, 17, 18, 26], which proposed features to automatically learn which warning is actionable. Due to space limit, we will not mention these studies here.

Heckman and Williams [8] conducted a systematic literature review of actionable warning identification techniques. They identified 21 studies and analyzed the identification approaches, the evaluation methodology, subject projects, etc. Allier et al. [1] proposed a framework to compare 6 warning ranking algorithms and identified the best algorithms to rank warnings. Similarly, Heckman and Williams [9] evaluated another 6 warning identification techniques. These studies are to review or evaluate the performance of the whole warning identification approach, rather than the effectiveness of every single feature, as what we do in this work.

Johnson et al. [10] investigated why developers are not widely using static analysis tools. They conducted reviews with 20 developers and found that false positives, and the way in which the warnings are presented are the main barriers. Beller et al. [3] conducted a large-scale evaluation about the state of the use of static analysis tools on open source software. Results revealed that their use are widespread, but not ubiquitous, and many projects typically do not enforce a strict policy on their use. Avgustinov et al. [2] presented an approach to track static analysis warnings over the history of a project, and further use the information to capture developer's characteristics. Smith et al. [19] investigated questions developers asked while diagnosing potential security vulnerabilities with static analysis. Thung et al. [21] studied to what extent could field defects be detected by the state-of-the-art static analysis tools.

[7]https://github.com/qhanam/Slicer

9 CONCLUSION

In this paper, we conduct a systematic experimental evaluation of all the available features for static analysis warning identification. Results demonstrate that there is a golden feature set (23 features) for warning classification. This finding can serve as a practical guideline for facilitating real-world warning identification.

Note that, this paper focuses on the relative effectiveness of the features, rather than explores the minimal subset of features that can achieve the best performance. Future work will conduct further exploration. Our evaluation is only conducted in terms of one SA tool (i.e., FindBugs), and the golden feature set might not perform well for other SA tools. Besides, although we employ a commonly-used feature selection method, other feature selection methods might come to a slight different golden feature set. Future work will also conduct evaluation for other SA tools and examine other feature selection methods to provide more solid conclusions and more guidelines.

10 ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under grant No.61602450, No.6143200, and China Scholarship Council.

REFERENCES

[1] Simon Allier, Nicolas Anquetil, Andre Hora, and Stephane Ducasse. 2012. A Framework to Compare Alert Ranking Algorithms. In *WCRE 2012*. 277–285.
[2] P. Avgustinov, A. I. Baars, A. S. Henriksen, G. Lavender, G. Menzel, O. d. Moor, M. Schafer, and J. Tibble. 2015. Tracking Static Analysis Violations over Time to Capture Developer Characteristics. In *ICSE 2015*. 437–447.
[3] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *SANER 2016*. 470–481.
[4] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing and Detecting Anti-patterns in the Logging Code. In *ICSE 2017*. 71–81.
[5] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. 2014. Finding Patterns in Static Analysis Alerts: Improving Actionable Alert Ranking. In *MSR 2014*. 152–161.
[6] Sarah Heckman and Laurie Williams. 2008. On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques. In *ESEM 2008*. 41–50.
[7] Sarah Heckman and Laurie Williams. 2009. A Model Building Process for Identifying Actionable Static Analysis Alerts. In *ICST 2009*. 161–170.
[8] Sarah Heckman and Laurie Williams. 2011. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology* 53, 4 (2011), 363 – 387.
[9] Sarah Heckman and Laurie Williams. 2013. A Comparative Evaluation of Static Analysis Actionable Alert Identification Techniques. In *PROMISE 2013*. 4:1–4:10.
[10] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don'T Software Developers Use Static Analysis Tools to Find Bugs?. In *ICSE 2013*. 672–681.
[11] Sunghun Kim and Michael D. Ernst. 2007. Prioritizing Warning Categories by Analyzing Software History. In *MSR 2007*. 27–31.
[12] Sunghun Kim and Michael D. Ernst. 2007. Which Warnings Should I Fix First?. In *FSE 2007*. 45–54.
[13] B. Kitchenham. 2004. *Procedures for Performing Systematic Reviews*. TR/SE-0401. Keele University.
[14] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. 2004. Correlation Exploitation in Error Ranking. In *ESEC-FSE 2004*. 83–93.
[15] Guangtai Liang, Ling Wu, Qian Wu, Qianxiang Wang, Tao Xie, and Hong Mei. 2010. Automatic Construction of an Effective Training Set for Prioritizing Static Analysis Warnings. In *ASE 2010*. 93–102.
[16] Foyzur Rahman, Sameer Khatri, Earl T Barr, and Premkumar Devanbu. 2014. Comparing static bug finders and statistical prediction. In *ICSE 2014*. 424–434.

[17] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. 2008. Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach. In *ICSE 2008*. 341–350.

[18] H. Shen, J. Fang, and J. Zhao. 2011. EFindBugs: Effective Error Ranking for FindBugs. In *ICT 2011*. 299–308.

[19] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2015. Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis. In *ESEC/FSE 2015*. 248–259.

[20] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2016. Automated Parameter Optimization of Classification Techniques for Defect Prediction Models. In *ICSE 2016*. 321–332.

[21] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. 2015. To what extent could we detect field defects? An extended empirical study of false negatives in static bug-finding tools. *Automated Software Engineering* 22, 4 (2015), 561–602.

[22] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. ALETHEIA: Improving the Usability of Static Security Analysis. In *CCS 2014*. 762–774.

[23] Ian H Witten and Eibe Frank. 2005. *Data Mining: Practical machine learning tools and techniques*.

[24] X. Ye, R. Bunescu, and C. Liu. 2016. Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-Grained Benchmark, and Feature Evaluation. *IEEE Transactions on Software Engineering* 42, 4 (2016), 379–402.

[25] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. 2007. An Empirical Study on Classification Methods for Alarms from a Bug-finding Static C Analyzer. *Inf. Process. Lett.* 102, 2-3 (April 2007), 118–123.

[26] U. Yuksel and H. Sozer. 2013. Automated Classification of Static Code Analysis Alerts: A Case Study. In *ICSM 2013*. 532–535.