WARNINGSGURU- Analysing historical commits, augmenting software bug prediction models with warnings and a user study

Louis-Philippe Quérel

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements for the
Degree of Master of Applied Science (Software Engineering) at
Concordia University
Montréal, Québec, Canada

April 2017

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Louis-Philippe Querel

Entitled: WarningsGuru - Analysing Historical Commits, Augmenting Software Bug Prediction Models with Warnings and a User Study

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. M. Kersten-Oertel Chair

Dr. E. Shihab Examiner

Dr. J. Paquet Examiner

Dr. P. Rigby Supervisor

Approved by _____
Chair of Department or Graduate Program Director

_____
Dean of Faculty

Date _____

Abstract

WARNINGSGURU- Analysing historical commits, augmenting software bug prediction models with warnings and a user study

Louis-Philippe Quérel

The detection of bugs in software is divided into two research fields. Static analysis report warnings at the line level and are often false positives. Statistical models use historical change measures to predict bugs in commits at a higher level. We developed a tool which combines both of these approaches.
Our tool analyses each commit of a project and identifies which commit introduced a warning. It processed over 45k commits, more then previous research. We propose an augmented bug model which includes static analysis measures which found that a twofold increase in the number of new warnings increases the odds of introducing a bug 1.5 times. Overall, our model accounts for 22% of the deviance which is an improvement over the 19.5% baseline. We demonstrate that we can use simple measure to predict new security warnings with a deviance explained of 30% and that recent development experience and more co-developers reduces the number of security warnings by 8%.
We perform a user study of developers who introduced new warnings in 37 projects. We found that 53% and 21% of warnings in Findbugs and Jlint respectively are useful. We analysed the time delta between the introduction and response of the developer to the notification of the warning. We hypothise that remembering the context of the change as an impact on the perceived usefulness given useful warnings had a median of 11.5 versus 23 days for non useful warnings

Acknowledgements

With the completion of a bachelor of software engineering in 2014 and now a master of applied science in software engineering, this concludes almost a decade of studies and research at Concordia University. Concordia has been a home and it will certainly not be one that I will forget any time soon.

I would like to express my highest and most sincere gratitude to my supervisor, Dr. Peter Rigby. I would not have undertaken a thesis if it were not for you. Without your support and guidance this research would not have been possible.

To my family, friends and colleagues. Without your moral support I would probably not have had the determination to finish this undertaking.

Contents

List of Figures

List of Tables

List of Abbreviations

CWE **C**ommon **W**eakness **E**numeration (MITRE Corporation, 2016)
JDK **J**ava **D**evelopment **K**it
TOIF **T**ool **O**utput **I**ntegration **F**ramework (KDM Analytics, 2016)

Chapter 1

Introduction

1.1 WarningsGuru

When developers have to deal with warnings for static code analysis tools there is often the recurring theme that it can be an overwhelming task. Static analysis tools have been reported to present too many warnings, with the majority of reported warnings being false positives (Ayewah et al., 2007). A perceived issue with the tools could hinder their adoption and use by developers (Johnson et al., 2013). Previous research has looked into augmenting static analysis tools by either improving their filtering (Nanda et al., 2010; Kim and Ernst, 2007) or their prioritisation (Kim and Ernst, 2007) of warnings which they report. These processes serve to reduce the number of warnings which are presented to the developer, but they did not consider the changes which introduced them. They also do not highlight warnings that have been recently introduced as part of the changes of the developers.

Static analysis tools for Java such as Findbugs and Jlint must be run on the build artifacts (*.class* files) of a project to operate correctly. We propose WarningsGuru, a tool which integrates source version control of the project, build tools and static analysis tools to build historical commits and identify which warnings originate from a given commit. WarningsGuru automatically builds and analyses all commits of a project. We prioritise warnings by highlighting those that have been introduced within a commit to inform the developer which warnings are caused by their changes. This reduces the number of warnings that need to be investigated.

Running the static analysis tools can be an expensive endeavor when all the files of a project need to be analysed. We limit the number of files that need to be analysed by targeting the ones which were modified as part of a change. We additionally devise a warnings recovery method for commits which fail to be built successfully which identifies the warnings that should be present on the modified files of failed commits. Using these methods we analyse over 50,000 commits on 37 software projects, with the oldest successfully built commit being from June 2006. This allows us to provide granular and timely details regarding the warnings of each commit of a project

Static analysis tools are used to reduce the occurrences of certain types of bugs in a software project. Another field of study which identifies bugs in software projects is statistical bug prediction. Statistical bug prediction uses measures which are mined from the history of the project such as code churn and developer experience from the project source version control or the results of test

suites to predict the occurrence of bugs based on the historical presence of bugs in the project. We propose the use of measures extracted with WARNINGSGURU to determine if warnings, which are also an historical component of the project, can be used as a bug prediction measures. We also propose that statistical models of warnings prediction could be used to reduce the number of commits to analyse using static analysis tools by prioritizing those which have a higher risk of introducing warnings.

In conclusion, with WARNINGSGURU's capability to identify the warnings as they are introduced, we assess the impact that these warnings have for developers. We undertake a user study to evaluate the developer's perceived usefulness of static analysis warnings by contacting the developers who introduced warnings as part of their commits. We evaluate if the tool which the warning originates from (Findbugs & Jlint) has an impact on the usefulness of the warning. Finally, by targeting the recency of warnings, we determine if the timeliness of the developer's knowledge of the warning has an impact on its perceived usefulness and propose a hypothesis for the result.

1.2 Structure of Thesis

This research thesis is broken down into three main chapters. These cover the creation of WARNINGSGURU and our subsequent use of it.

In Chapter 2 we describe the architecture of WARNINGSGURU, a tool which we developed which is capable of building and running static analysis on the historical commits of projects. We discuss the issues which we faced in its creation and the techniques that were employed to resolve them. We demonstrate WARNINGSGURU by building 45 thousand commits of eight projects and determine which commits introduce warnings into the project.

In Chapter 3 we use the data generated by WARNINGSGURU to improve statistical bug prediction models. We then build warnings prediction models which can predict the occurrence of warnings in the commits of a project using common bug prediction measures.

In Chapter 4 we perform a user study of newly introduced warnings identified by WARNINGSGURU to assess their perceived usefulness by developers. We evaluate and compare the results of our two static analysis tools to assess their usefulness. We determine the impact that the delay of notification of a warning as on its perceived usefulness by the developer.

Finally, in Chapter 5 we provide a final review of the findings of this thesis.

Chapter 2

WARNINGSGURU: Architecture & Data

2.1 Introduction

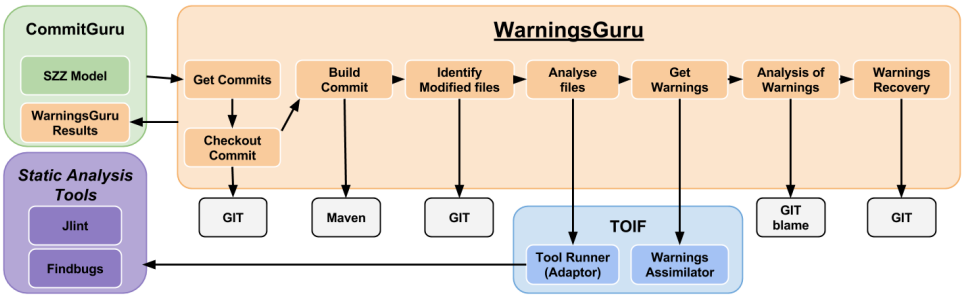
Static code analysis tools find defects by examining the code, data-flow and control-flow for problematic patterns of code. To make static code analysis tractable in practice, tools such as FindBugs and JLint make simplifications and abstractions that leads to a large number of warnings. Many of these can be false positives, including trivial and unlikely warnings (Ayewah et al., 2007; Beller et al., 2016). The advantage of static analysis is that the warnings are specific, *e.g.* a null pointer on a specific source code line. The disadvantage is the overwhelming number of reported warnings and a disconnect between field defects and warnings (Couto et al., 2013).

We developed a tool called WARNINGSGURU which integrates source version control, build tools and static analysis to build and run static analysis on thousands of historical commits. Using WARNINGSGURU we identify from which commit a warning originates from to identify which warnings are new to a commit. We also develop a technique which traces the warning introducing commit where the commit failed to build. This allows us to retroactively assign warnings to failed builds using WARNINGSGURU.

We integrate the results of WARNINGSGURU into the interface of COMMITGURU (Rosen, Grawi, and Shihab, 2015) where we present the new and historical warnings which WARNINGSGURU identified. COMMITGURU is a tool which extracts commit attributes and creates statistical bug prediction models on these projects. The warnings are associated to the file where the warning is reported and it is linked to the code diff on GitHub where the developer can observe the change which introduced the warning. A new warning is defined as a warning that is on a line which was last modified by the same commit which was analysed. We also indicate with each commit if WARNINGSGURU was able to successfully build and analyse the commit.

This chapter is structured as follows. In Section 2.2 we provide the architecture of our tool and provide screenshots showing how developers can view new warnings on commits which COMMITGURU as determine is buggy. In Section 2.3, we describe how we successfully build over 33k commits and retroactively assign warnings on the 12k commits that do not build. In Section 2.4 we evaluate the results from WARNINGSGURU. Section 2.5 reviews the risks to validity and

FIGURE 2.1: WARNINGSGURU pipeline architecture



WARNINGSGURU integrates COMMITGURU, Git, Maven and TOIF to build and analyse historical commits. Refer to Appendix A for larger figure

Section 2.6 looks into the prior work that has been done in the field of static analysis of software projects. Finally, Section 2.7 provides the concluding points on the functions of WARNINGSGURU.

In addition, we make the tool available. The source code is publicly available on GitHub: `https://www.github.com/louisq/warningsguru`

2.2 WARNINGSGURU Features and Architecture

We give an overview of the features of WARNINGSGURU as shown in the pipeline in Figure 2.1. We then present the results from WARNINGSGURU in a modified interface based on COMMITGURU.

2.2.1 WARNINGSGURU Pipeline Architecture

WARNINGSGURU builds on an existing statistical bug modeling tool, COMMITGURU (Rosen, Grawi, and Shihab, 2015); static analysis tools, including FindBugs (University of Maryland, 2017) with a security bug detection plugin (Philippe Arteau, 2017) and Jlint (Cyrille Artho, 2017); and a static analysis warnings integration tool, TOIF (KDM Analytics, 2016). It targets projects written in the Java programming language which are using Maven as their build tool.

The following sections below describes the steps of the pipeline as defined in Figure 2.1.

COMMITGURU COMMITGURU is used to download the repositories of the projects and to maintain their daily updates. We are also using CommitGuru to obtain the list of commits and the measures attributable to the commits of the project.

Get Commits WARNINGSGURU obtains the list of commits which COMMITGURU as extracted that is as not analysed yet. This includes historical commits and newer commits as COMMITGURU updates the repositories. WARNINGSGURU analyses commits incrementally from the newest commit which as not been analysed yet.

Checkout Commit By using Git, WARNINGSGURU checkouts the repository of the project to the commit which is being analysed.

Build Commit WARNINGSGURU then analyses the file structure of the commit to determine if the commit supports the compatible build system: Apache Maven. This is achieved by confirming the presence of a Maven POM configuration file (The Apache Software Foundation, 2016). If the commit is compatible it attempts to build the commit.

Identify Modified Files Whether the Maven built of the commit succeeded or failed, WARNINGSGURU determines which class files originate from modified files of the targeted commit. This is done to reduce the number of files which WARNINGSGURU analyses in subsequent steps. This reduces the number of files which need to be analysed by WARNINGSGURU and doesn't require each commit of the project to be completely analysed each time.

Analyse Files The use of TOIF (KDM Analytics, 2016) by WARNINGSGURU allows for the execution of different static code analysis tools without requiring the integration of each individual tool into the pipeline. WARNINGSGURU runs Findbugs and Jlint through TOIF on the modified class files which it has identified.

Get Warnings WARNINGSGURU extract the list of warnings associated from the files which were analysed by the static analysis tools. TOIF augment these results with CWE (MITRE Corporation, 2016) and SFP (KDM Analytics, 2016) which are respectively a warning type classification and security-related type classification. These warnings do not indicate which commit they originate from and whether the analysed commit introduced the warnings.

Analysis of Warnings Using Git blame, WARNINGSGURU identifies the origin of each line to determine which commit last modified the line of the warning. If the line was modified by the current commit then it is determined that the warning is new. WARNINGSGURU is able to determine the origin of a warning based on information of the last modification of the line it is assigned to.

Warnings Recovery Since we trace the introduction of each line in a file, we can retroactively identify warnings to commits which did not build. WARNINGSGURU determines the warnings which should be present in the modified files of a commit and can identify the historical and new warnings that would be present even if the commit fails to build.

WARNINGSGURU Results The warnings are then presented as part of the interface which is described by the next section.

We use the tool TOIF in WARNINGSGURU given that this research was funded by a grant which was supported by the company that developed it. TOIF currently primarily supports static code analysis tools for the Java and C/C++

programming languages. Given that the author had negligible experience with C/C++ languages it was decided to cover the Java programming language as part of this study. By extension, TOIF supports the Findbugs and Jlint static code analysis tools for Java which is why these tools were selected. While it is possible to add additional static analysis tools to TOIF such as PMD, this would have required the creation of the warnings classification mapping for warnings of these new tools. It was decided that the results from Findbugs and Jlint would be sufficient to evaluate the warnings extraction method of WARNINGSGURU. We also use the security classification of the warnings as were assessing the effectiveness of security warnings as part of the research grant.

2.2.2 WARNINGSGURU Interface

A selected portion of the results extracted from WARNINGSGURU is presented in our modified interface of COMMITGURU which integrates these results. The features are illustrated by Figure 2.2 and are listed below.

1. WARNINGSGURU indicates for each commit the total number of warnings and new warnings and COMMITGURU's commits which have been determined to be buggy. It also indicates the status of WARNINGSGURU on a commit by the use of the left most coloured tag. For example green means that the commit was built and red means that the build was not successful.

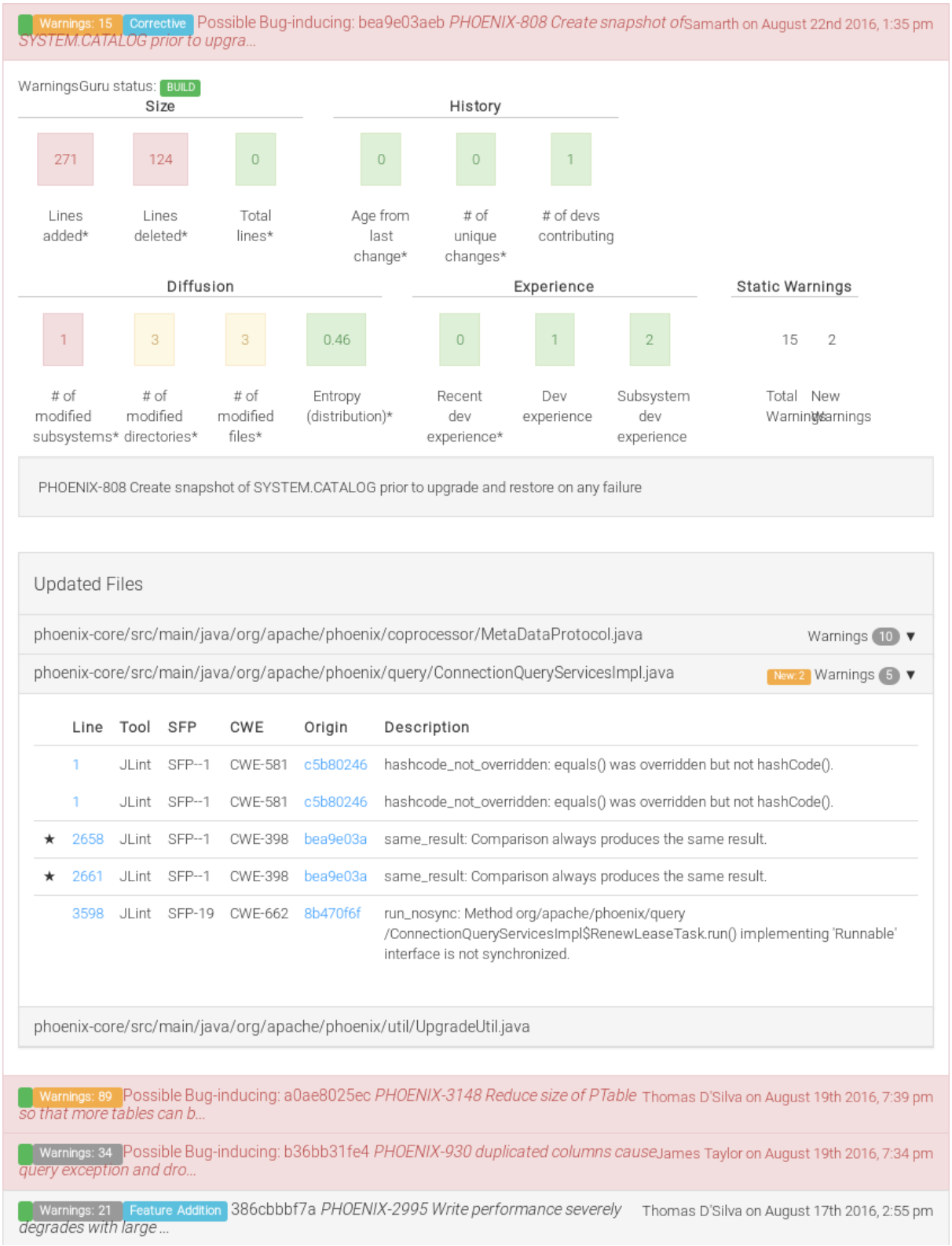
2. WARNINGSGURU indicates the number of existing and new warnings associated with each file for a commit and the measures that indicate the risk in COMMITGURU's model of a commit introducing a bug.

3. The warnings per line reported by FindBugs and JLint are displayed when a file is selected. WARNINGSGURU reports the commit in which the warning first appeared and the line which it is presently on. The star (★) next to the warning indicates that this is a new warning that was introduced as part of the commit. Warnings which do not have a star originate from historical commits to the modified file.

4. Each warning is clickable, which directs the developer to the highlighted problematic line on GitHub. By clicking the commit hash of the originating commit, the developer will instead be taken to the original line of the commit on GitHub which introduced the warning (Figure 2.3).

5. New projects can be added simply by submitting the Git URL of a Maven project hosted on GitHub to WARNINGSGURU. Once a project is added new commits will be continuously built and analysed by WARNINGSGURU (not shown in the figure).

FIGURE 2.2: Integration of WARNINGSGURU in modified COMMITGURU interface

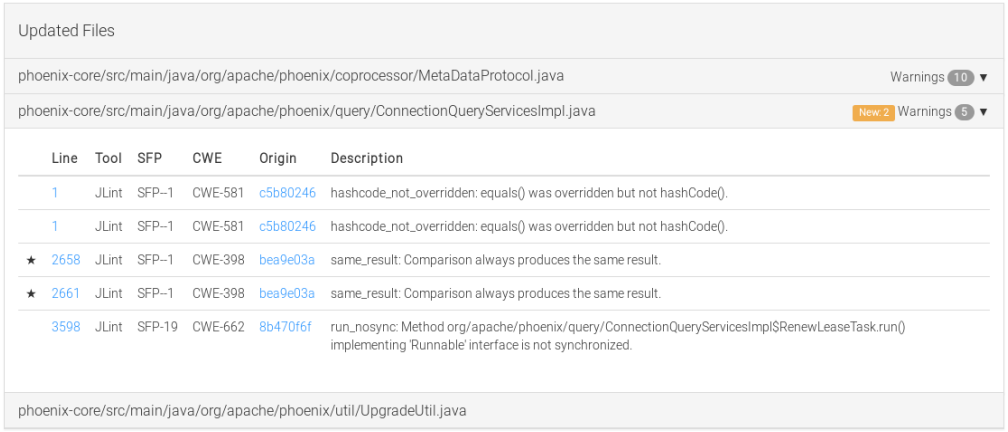


Each commit lists the files which were modified as part of the selected commit with the warnings associated to each individual files and the risk predictors for the commit. The individual warnings are shown with links to their originating line on GitHub. A star indicates a new warning, given that the line to which the warning is associated was last modified as part of the commit which was analysed.

FIGURE 2.3: Presenting line location of warning in GitHub

For each warning presented, WARNINGSGURU provides a link for the line in the commit which the warning is presented and the commit where the warning was introduced as shown in Figure 2.2. The line which the warning is associated to is highlighted in a snapshot of the file of the respective commits as part GitHub. This provides context for the warning in relation to the code of the project.

FIGURE 2.4: Historical and New Warnings



An example of the warnings as reported by WARNINGSGURU for a file of commit *bea9e03a*. Warnings which are on a line which was last modified by *bea9e03a*, the current commit, are identified as new warning by the use of the inline star (★).

2.2.3 Identification of New Warnings

We defined new warnings to be warnings which are located on lines that were last modified by the same commit to which the warnings were reported. Figure 2.4 shows and example of the warnings which WARNINGSGURU identified on the file *phoenix-core/src/main/java/org/apache/phoenix/query/- ConnectionQueryServicesImpl.java* of the Phoenix commit *bea9e03a*. In this example we have 5 warnings which are located on lines which were last modified by the commits *c5b80246*, *c5b80246*, *bea9e03a*, *bea9e03a* and *8b470f6f* respectively as defined by the *Origin* column in Figure 2.4. These information is obtained using Git blame for the warnings extracted in each commit . The Git blame process and parameters used are covered as part of Section 2.3.3.

Given that lines of the 3rd and 4th warnings are determined to have been last modified by the same commit, *bea9e03a*, these warnings are identified as new warnings of the commit. New warnings are indicated with an inline star (★) as presented in Figure 2.4. The lines of the 3 other warnings were last modified by other commits which preceded the commit *bea9e03a*. These warnings are therefore historical warnings of the project which already existed before the analysis of the current commit.

2.3 Data: Building & Analyzing Thousands of Commits

Recent works that combine static analysis with statistical bug models include only a small number of project snapshots and do not provide a developer tool (Rahman et al., 2014; Tang et al., 2015). For example, Rahman *et al.* study between 5 and 8 release for five open source projects for a total of 34 releases. They state that the effort to build and run JLint and FindBugs on the 34 versions took six person-months of effort. Tang et al., 2015 studied 3 and 5 releases of 2 projects for a total of 8 releases. Nanda et al., 2010 created a private tool which ran static analysis tools on commits, but did not build current or historical commits.

In contrast, we process 45,949 commits an increase of 1,351 times as many versions compared to Rahman *et al.*. By processing all the commits of a project, WARNINGSGURU gives precise and timely information about when a static analysis warning *first* appeared, simplifying future comparisons of statistical and static bug predictions and giving developers up-to-date information.

To allow WARNINGSGURU to be able to build and run static analytic tools on older commits, we targeted projects which had the following characteristics:

Contained Build management: Apache Maven provides a self-contained build process. While there are other build tools which are available for Java such as Ant, Gradle and SBT which perform the same operation, they are either not as broadly used or are more recent then the forward compatible POM format used by Maven which was introduced in 2005. The motivation behind this decision is that it allows WARNINGSGURU to support projects with commits which are up to 12 years old at the time of the research, allowing WARNINGSGURU to analyse older projects without needing to support multiple build tools.

Dependency management: Software projects use third party libraries to allow for reuse of code and functionality. A build will fail if these libraries are no longer available. Maven uses centralized repositories of libraries where historical versions of these libraries are preserved and distributed. As long as the libraries required for a commit are still distributed on these repositories it is possible to obtain all of the dependencies required by the commit configuration. We are therefore capable of building older commits in the majority of cases without needing manual intervention.

Similarity to commercial software: The Apache Software Foundation which as been founded in 1999 and as been managing over 350 open source projects which follows processes of software engineering and management. The selection of their project allow us to analyze large successful software projects.

2.3.1 Building

Java static analysis tools typically require compiled files of a project to run, *e.g.* class files. To build each historical commit, WARNINGSGURU requires the commit to contain a Maven POM file (The Apache Software Foundation, 2016). A POM file stores the build configurations of the project including the required versions of dependencies and version of Java allowing us to build historical versions of the project.

Researchers and developers can add new projects to WARNINGSGURU provided that a Maven POM file exists. The processing time depends on the project build time and the number of files that changed. Multiple instances of WARNINGSGURU can be run in parallel to increase the number of commits which can be analysed concurrently.

Building historical commits is a complicated process which required a multifaceted approach to ensure a maximal number of successful commits are built. The following subsections describe some of the issues which were identified in the development of WARNINGSGURU and how we address them if a solution was available.

Evolution of Build Tools

The build tools which the projects are using have also been changing as the respective projects progress. We observed that some were not specifying in their POM files the version of Java which they were targeting to be used as part of their build. While newer versions of the Java Development Kit (JDK) should be able to build code which was meant for older versions of Java, the lack of targeting resulted in incompatibilities which caused builds to fail. WARNINGSGURU implements a mechanism to override the version of the JDK used for building based on the date of a commit to allow to build older commits which don't support the newer versions of Java. Maven releases also support specific major releases of the JDK. Given that we override the JDK, WARNINGSGURU also provides a mechanism to override the version of Maven which ensures that a compatible version of the build tool is employed to perform the build.

The parameters used by the tools can also evolve with subsequent releases. Some of the projects have their build configured to fail where a test on the

application is unsuccessful. As we are only building to obtain the build artifacts, we are not interested in the results of the tests which would significantly reduce the performance of WARNINGSGURU. We therefore disable their execution as part of the build process. However the parameter which is used by Maven *e.g.*-DskipTests was previously different which requires the support of multiple test exclusion parameters to ensure that the tests are not executed.

Projects were also observed to have previously used different build tools in their history such as Ant and the configuration format of Maven 1 which is incompatible with the versions of Maven used in WARNINGSGURU. To be capable to build even older commits would require the support of additional build tools which have their own flows.

Missing Dependencies

While Maven provide mechanisms to manage the dependencies, it is still possible that a build fails due to a missing project configured dependency. In addition to the default dependency repository (Maven Central), some of the projects configure additional repositories that contain dependencies which might not be available in Maven Central. These additional repositories might no longer be operational which would result in a missing dependency that results in a failing build. To mitigate this issue we have added additional repositories to our Maven instance that include dependencies that are not present in Maven Central.

Other projects have been observed to use 'SNAPSHOT' dependencies which have become no longer available. Snapshot dependencies are versions of a library which are distributed before the formal release of a library version. Snapshots are then deleted when the completed library is released. Snapshot versions have a different identifier which distinguish them from the final release of a version and when a build attempts to obtain one that has been deleted it will cause it to fail. Since the final release is a close relative of the snapshot, the final release should be a stabilized version of the snapshot version. A solution is to substitute snapshot library with the final release one. We have completed this step manually with some success, however it was only performed a limited number of times and additional work would be required to determine how to automate this process.

While the approaches above may resolve some of the issues, there were still some dependencies which could not be resolved which resulted in failed builds

Complex Build Configurations

Maven is a build tool that permits the use of third party plugins to extend its functionality. Some of these plugins integrate with other tools and services which need to be available on the build environment. The availability of these tools may cause issues to automate the build process where they are required. We had to reject some projects from this study due to their additional external tools that they required on the environment to be built. An example of this is Apache Falcon which also required node.js to be configured on the environment.

TABLE 2.1: Selected Projects Time Frame

Project	First Commit	Last Commit	Commits
Commons-lang	2006-11-11	2017-02-04	3314
Hadoop	2011-08-02	2017-02-03	14458
Ignite	2015-05-29	2017-02-03	4368
Kylin	2014-10-02	2017-02-04	5749
Phoenix	2014-01-27	2017-02-03	1892
Ranger	2014-01-27	2017-01-06	1913
Tika	2007-03-31	2017-02-02	3345
Wicket	2006-06-29	2017-02-04	10910
		Total:	45949

An earlier version of WARNINGSGURU integrated itself in the build process through a Maven plugin. The change to the build process, while minimal, would cause a certain number of builds to fail due to incompatibilities with the plugin which we used and the ones that the projects had configured to use. The current pipeline as presented in Figure 2.1 does not use this approach and instead performs the analysis following the build. This reduces the risk that WARNINGSGURU is the cause of the build failure by preventing the tool from directly modifying the build process.

Additional parameters can also be provided to Maven as part of the execution command. Some of the projects require additional memory to be capable of managing the complexity of the build process. This information is usually provided as part of the project's readme. In order to allow these builds to succeed, additional memory is allocated as part of the execution which WARNINGSGURU performs by default. This reduces the risk that the build would fail due to insufficient memory.

Projects Build Results

As part of this research study, we analysed each commit from the Apache project: Commons-lang, Hadoop, Ignite, Kylin, Phoenix, Tika, Ranger and Wicket. These projects contain between 1900 to 20000 commits over multiple years (see Table 2.1). Each project is written primarily in Java and uses the Apache Maven build tool to manage their build configuration and they are available on GitHub. We briefly describe each project.

Commons-lang is a library which provides additional utilities to the core Java classes. Hadoop is a distributed computing and storage platform. Ignite is an in memory computing platform. Kylin is a distributed analytical engine which interacts with solutions such as Hadoop. Phoenix is a library that provides SQL support for non-SQL databases. Tika is a tool which extracts metadata from files which can be used for indexing. Ranger is a monitoring and security utility for Hadoop. Wicket is a web framework to build Java server side based services.

TABLE 2.2: Build Results of Commits in Selected Projects

Project	Commits	Success	Failure	Build Success
Commons-lang	3314	3123	191	94.2 %
Hadoop	14458	9360	5098	64.7 %
Ignite	4368	3606	762	82.6 %
Kylin	5749	5084	665	88.4 %
Phoenix	1892	1818	74	96.1 %
Ranger	1913	961	952	50.1 %
Tika	3345	3166	179	94.7 %
Wicket	10910	6164	4746	56.5 %
Total:	45949	33282	12667	72.4 %

TABLE 2.3: Commits warnings including recovered warnings

	Commits	Percentage
Total	45949	-
With warnings	26898	58.5%
With new warnings	5881	12.8%
Total successful build	33282	72.4%
Successful with warnings	19553	58.7%
Successful with new warnings	4387	13.2%
Total failed build	12667	27.6%
Failed with warnings	7345	58.0%
Failed with new warnings	1494	11.8%

2.3.2 Static Analysis Integration - TOIF

Each static analysis tool has its own execution flow and method of reporting warnings. We use TOIF (KDM Analytics, 2016), an open source framework developed by KDM Analytics, to integrate static analytics tools into the pipeline of WARNINGSGURU. TOIF provides a common execution API for the static analysis tools and parses their results to convert them into a common format. We currently run JLint and FindBugs, which are Java static analysis tools on the commits which we build.

JLint and FindBugs provide more than just code linting functionality, they also identify some types of weaknesses such as index out of bounds errors in arrays and invalid comparisons. We are also using the find security bugs Philippe Arteau, 2017 plugin for FindBugs which provides additional warnings for issues such as an improper use of encryption and potential injection vectors which are not identified by the standard tool.

TOIF enriches the warnings by mapping them to software security warnings classifications including the *common weakness enumeration (CWE)* (MITRE Corporation, 2016) and *software fault pattern (SFP)* (KDM Analytics, 2016) categories. This mapping is security-focused, allowing developers and future researchers interested in security to ignore warnings that rarely lead to security problems. Figure 2.2 shows the integrated and additional warning fields in a GUI in WARNINGSGURU.

2.3.3 Version Control System - Git

Version control system are used to manage the source code of software projects where incremental changes are stored with auditing of who and when a change occurred. WARNINGSGURU uses Git to retrieve the state of the system at each commit. The system state at each commit is then analysed for the following three purposes.

First, we walk the Git DAG to checkout the state of the system at each commit. WARNINGSGURU then builds and runs static analysis tools on each commit.

Second, running static analysis tool is computationally expensive. We use Git to identify and only perform the static analysis on the files which have been modified as part of the commit. Following the build, we can use these files to determine which of the compiled files need to be analysed using the static analysis tools to extract the warnings.

Third, we use Git blame to determine the commit in which a line in a file was last modified. This allows us to determine the historical commit in which the warning was introduced. We use the following git command on the lines which have warnings attributed to them in a file:

```
git blame -lnswfMMMCCC {line_numbers} {file_path}
```

This command allows WARNINGSGURU to differentiate between the lines and determine which warnings are new in a commit. The *-l* parameter shows the long commit hash. The *-n* determines the line number in the original commit

TABLE 2.4: Example of git blame results

Origin commit	Origin file path	Origin line #	Current line #
^c5b8024	src/.../AlterTableTest.java	40	41
d3aba78a	pho.../AlterTableIT.java	43	42
1510cd2d	pho.../AlterTableIT.java	42	43
6cc17278	pho.../AlterTableIT.java	46	44

Example of results obtained through the use of the git blame command in WARNINGSGURU. Some of these values have been truncated for presentation purposes such as the line content which is not presented or the origin commit and file path which have been truncated.

where the line was last modified. The *-s* suppresses the author name and commit time stamp as WARNINGSGURU already obtains this information from COMMITGURU. The *-w* parameter ignore white space changes and is made to identify the origin of a line. The *-f* shows the file path when the line was last modified. Finally the *-MMMCCC* allows for the tracking of movement of files and lines and the copying of these within a project.

An example of the result is presented in Table 2.4. The origin commit is the commit hash of the commit which last modified the line. Where a commit hash starts with a ^, it implies that this line originates from the first git commit of the project. The origin file path is the file path of the line when it was last modified as part of the specified commit. The origin line number is the line number of the modified line when it was modified. Finally the current line number is the line number of the line at the commit snapshot where the command was ran. This is used to determine which warnings have been introduced by the commit and to differentiate between warnings in the process of the warning recovery. Figure 2.2 shows how developers can click on the “Origin" commit of older warnings or view the lines with new “Starred” warnings.

The measures obtained by COMMITGURU are also obtained from the source version control repository in order to build its statistical risk model.

2.3.4 Warnings Recovery

Where a commit fails to build successfully, it might not be possible to obtain the warnings from all modified files using the static analysis tools. A failure of a commit to build does not imply that no build artifacts are generated. WARNINGSGURU runs the static analysis tool on the successfully built artifacts of modified files and extracts the warnings from these files. However we cannot run the static analysis tools on build artifacts which do not exists as they were not compiled. The new warnings which are introduced as part of the commit can be obtained from subsequent analysis of its modified files in subsequent commits, but it would not specify which warnings are present on the commit which failed to build. We developed a warnings recovery process which identifies the warnings present in the modified files of a commit.

For each file which is modified in the commit, we identify all of the commits which previously modified the specified file. This provides us with the history

of the file, including rename and moves to allow us to uniquely identify files. We can then determine that two files are the same and obtain the full list of warnings which WARNINGSGURU as identified for this file throughout its history. If newer commits have been previously analysed, we can also determine which more recent updates of the modified files exist. This process however does not distinguish between warnings and it is not possible to determine that the warnings are unique.

We use the Git blame functionality to obtain the commit hash, file path and line number origin information associated to the line of a warning to determine the origin of the warning. This information is used to distinguish between warnings and uniquely identify them. By using the combination of origin commit hash, file path, line number and warning details as a unique identifier, we can uniquely identify that two warnings between commits are the same. A line can have more then one warning, but their details will be different which we use to distinguish between them. Using this approach we are able to uniquely identify which warnings are the same between commits.

When this approach is applied to the modified files of a commit, we can use the same approach to determine which warnings will be present on these files. For each modified file we obtain the origin information concerning each lines of these files. We then compare the origins of the lines to the origins of warnings which WARNINGSGURU as previously identified on the file. The list is filtered and we obtain the list of warnings that would be present on the commit.

The warning recovery process is dependent on the analysis of the project by WARNINGSGURU. We can only identify warnings which have previously been observed by WARNINGSGURU on the specified file. We therefore run the warning recovery following the completion of the analysis of a batch of commits.

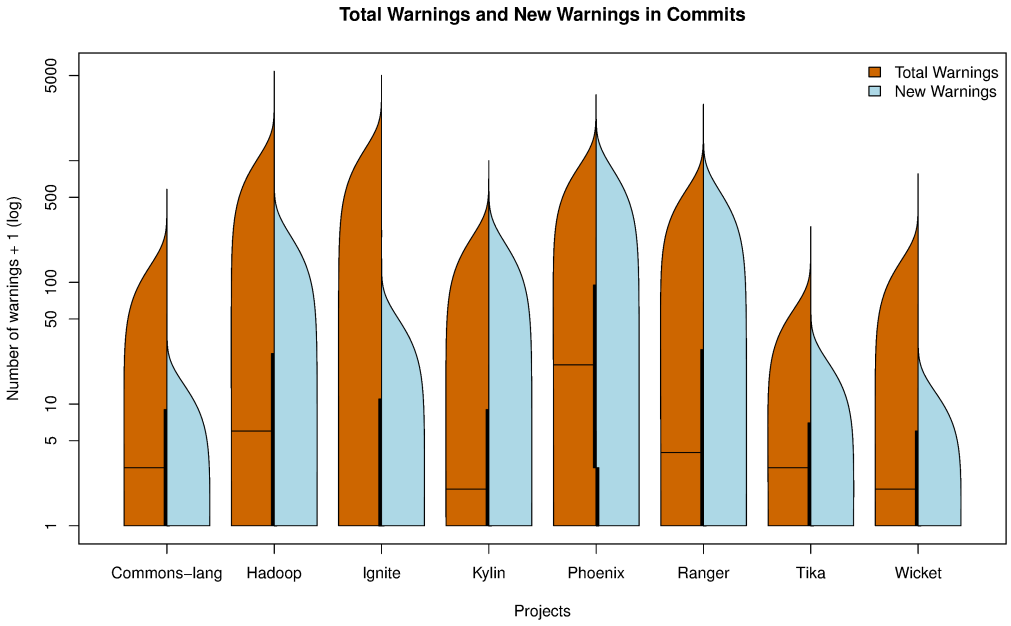
2.3.5 Statistical Models with COMMITGURU

Statistical models indicate when a change may introduce a bug by predicting when a commit or file is prone to being buggy using historical measures such as churn, entropy of change and the experience of the developer (Kamei et al., 2013). By being able to identify bug fixing commits from either an issue tracker or commit message it is possible to determine which commits might have introduced the faulty change. The measures from these candidate buggy commits are then used to identify other commits which may also be bug-introducing based on historical patterns.

COMMITGURU implements the SZZ/ASZZ algorithm (Kamei et al., 2013). Bug fixing commits are identified based on keywords in their commit message. Using Git blame, the fixing commit is traced back to the commit that last changed the fixed lines. The lines that have been changed are deemed to be bug-introducing. Measures are extracted at each commit and used as predictors in a logistic regression. This logistic regression model is used to predict whether or not a commit is likely to introduce a bug, *i.e.* is risky.

In Figure 2.2 we see for each commit whether the it introduces a bug based on the results of the SZZ algorithm. We also see which measures contribute

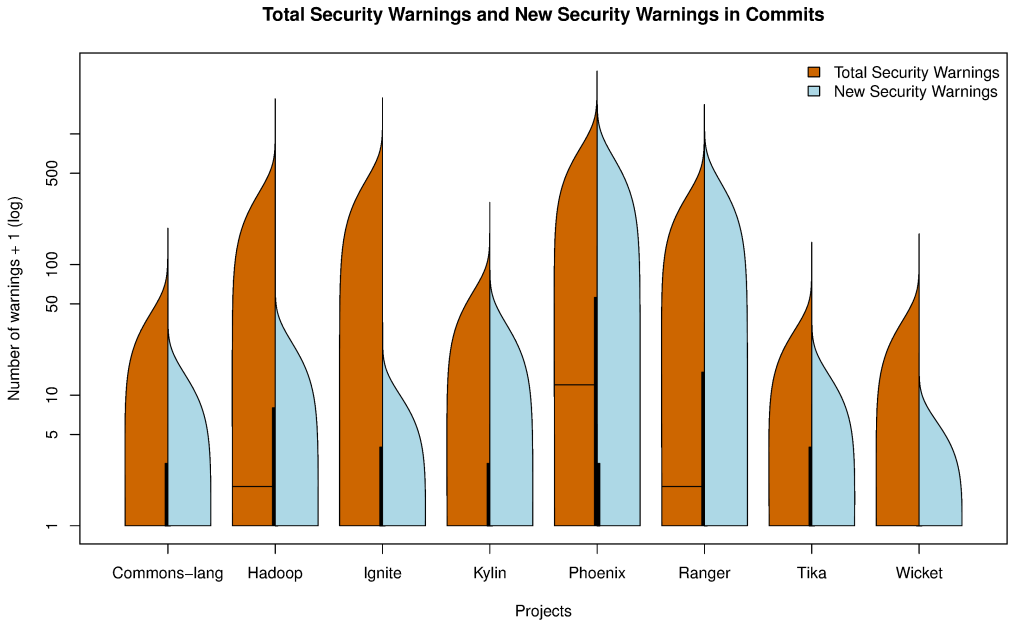
FIGURE 2.5: Distribution of warnings and new warnings



While a median number of commits contain warnings, few of these introduce new warnings. This violin plot represents the distribution of the datasets per project for the total number of warnings and new warnings per modified files of a commit. The peak is the maximum number of a type of warnings which are present in a commit. The horizontal line present in some of the plots is the median of the dataset of a project for a type of warning as define by the side of the plot it is on. Except for Ignite, all other projects have a median total number of warnings per commit which is above 0. The thicker lines between the median lines represent the second and third quartiles of the dataset where the second is below and the third is above the median line. The first and fourth quartile are presented on the plot as the portion before and after the second and third quartiles which do not have a thicker line. This is observable in the total number of warnings in the project Phoenix where all four quartiles are presented. Where the median line is not visible, the third quartile may be observed with the fourth quartile of the dataset on the violin plot.

additional risk. While this allows COMMITGURU to determine which commits may be the more risky, it does not give additional details of where the issue might be in the commit as the predictions are not at the line level. For example, Figure 2.2 shows that the model predicts a change to be risky because many new lines are being added in the commit. WARNINGSGURU supplements this knowledge with specific static analysis warnings that have been introduced in the change.

FIGURE 2.6: Distribution of security-related warnings and new security-related warnings



As opposed to warnings, fewer commits have security warnings and new security warnings. This violin plot represents the distribution of the datasets per project for total number of security warnings and new security warnings per modified files of a commit. Only Hadoop, Phoenix and Ranger have a median total number of security warnings per commit which is above 0.

2.4 Descriptive Statistics

We built and ran the static analysis on the commits of the eight Apache projects using WARNINGSGURU. The results of this are presented in Table 2.1. We were capable of successfully building over 33,000 commits with a success rate with a range of 94% to 50% per project (Table 2.2). This represents over 685,000 warnings that were collected through the analysis of the commits by WARNINGSGURU. The historical and new warnings can then be viewed on each commit in the interface of WARNINGSGURU as shown in Figure 2.2.

The warnings recovery process identified warnings on an additional 7345 commits which failed to build, of which 1494 had new warnings. By including these warnings identified through the recovery we obtain 941,000 warnings. We determine that the proportion of commits with warnings in failed builds is proportional to the occurrence of warnings in successfully built commits. In Table 2.3, we observe that 58.7% and 58.0% of commits have warnings in successfully and failed builds respectively. The differences between commits having new warnings for build versus failure is 13.2% and 11.8% respectively. Due to the similarity, the recovery of warnings in failed builds is therefore comparable to the occurrence which was obtained through the analysis.

The results showed that 58.5% percent of commits contain warnings in

the files which were modified and that 12.8% percent of commits introduced new warnings. There is a high of 18.0% on Ranger and a low of 3.6% on Wicket for percentage of commits with new warnings. We also observed that 7.3% of commits which had new warnings were deemed to be risky based by COMMITGURU. Commits with security warnings accounted for a mean of 40.7% of commits having a security warning and 6.8% having a new security warnings. Similar distributions were also observed with Ranger having a high of 12.3% and Wicket a low of 0.7% for commits with new security warnings. While a significant number of commits have warnings associated with them, substantially fewer have new ones making investigations of these by developers more manageable.

Analyzing the results for Tika, 53.9% of commits modified a file which contained a warning. Given that warnings is the only measure which is present in over 50% of commits it is the only one which has a median which is of 2.0 warnings per commit. We have a mean of 5.8 and a max of 286 warnings per commit. Of these, 14.4% of commits introduced a new warning which represent a mean of 0.6 and a max of 104 new warnings per commit. A subset of warnings are security warnings which were identified in 37.8% of commits. These account for a mean of 2.5 and a max of 147 security warnings per commit. Finally, new security warnings were found in 9.0% of commits. There is a mean of 0.3 and a max of 70 new security warnings per commit.

The observations associated to warnings also occurs in the other projects as presented in Figure 2.5. We observed similar trends in with security warnings which is presented in Figure 2.6.

2.5 Threats to Validity

All eight projects were written in the Java programming language, are part of the Apache Software Foundation, and used Maven as their build tool. The Apache Foundation includes a large number of representative projects and we have selected a range of projects that represent good software development practices.

Building historical commits is not a simple task. Missing dependencies, environment miss-configuration and incorrect version of the build tools are issues which might result in older builds bring more likely to fail. In Section 2.3.1 we developed techniques, such as trying older Java environments based on the commit date to minimize these risks. There are limitations to these efforts, for which our warnings recovery process compensates.

Static analysis tools cannot recover all of the warnings on failed builds. There is a risk of lost warnings which we mitigate by retroactively identifying the warnings that modified files in a failed build would introduce through the recovery process.

For projects or individuals which are independently using static code analysis tools it is a possibility that there are warnings which would have been in the commit which the developer as already addressed. However, given that WARNINGSGURU uses a combination of tools with the additional security plugin for Findbugs it is likely that it will identify other type of warnings. All projects

which we analysed had warnings and we are therefore still able to extract some from them. Also, the use of static analysis tools by these individuals would demonstrate that these tools are useful to them and it does not preclude research into the improvement of these tools.

While the warning recovery process is capable of obtaining warnings for commits which failed to build, there is a possibility that it might be missing some warnings and including others which would not have been on a file if the static analysis tools had been capable of running. Given that the recovery process is dependent on preceding and subsequent analysis of the modified file it is possible that the warning might never be identified if the file is never analysed again. More warnings might be present where the static analysis tool attributes multiple warnings to one line which we cannot differentiate given that they are all attributed to one line, but the warning might have originated from another part of the code.

2.6 Related Works

Many previous studies have investigated the use of static analysis tools (Yi et al., 2007; Kamei et al., 2013; Rahman et al., 2014; Tang et al., 2015; Wong et al., 2016). Some of the studies investigated the incremental analysis of snapshots of a project. Kim and Ernst, 2007 built and ran static analysis tools on over 4500 commits. They proposed a solution to improve the prioritization of warnings based on historical fixes of warnings to determine which categories of warnings are predominantly addressed by developers. Spacco, Hovemeyer, and Pugh, 2006 analysed 116 incremental releases of the JDK with Findbugs where they tracked warnings between each releases. They mention that collisions in warnings might cause issue in the tracking of warnings. This is attributable to them not the line number of the warning or its offset which our warning recovery process to track the warnings between releases. Bevan et al., 2005 developed Kenyon, a tool that allows for incremental analysis of software projects through its collection of change data between snapshots. It was however not capable of building historical commits. Nanda et al., 2010 also created a private tool which ran static analysis tools on commits, but did not build current or historical commits. Tang et al., 2015 studied 3 and 5 releases of 2 projects for a total of 8 releases.

Another aspect that has been studied is the prioritisation and filtering of warnings. Kim and Ernst, 2007 determined that between 6-9% of static analysis warnings were subsequently fixed as part of subsequent commits. They proposed a solution to improve the prioritization of warnings based on historical fixes of warnings to determine which categories of warnings are predominantly addressed by developers. Nanda et al., 2010 determine that the results from different static analysis should be merged and filtered when presented to developers. They developed a tool that runs static analysis tools on snapshots of a project and can report which warnings originate from the current change in addition to filtering the results. WARNINGSGURU supports for the merging of the results of static analysis tools and for the prioritisation of new warnings in a commit.

The results of static analysis tools have been applied to the identification of bugs in software projects. Zheng et al., 2006 determined that static analysis tools are complementary to other fault detection techniques. They can be used to determine the risk factor for testing purposes and the warnings that are associated to some security vulnerabilities. The warnings which WARNINGSGURU provide are associated to security categories. Wedyan, Alrmuny, and Bieman, 2009 identified that fewer than 3% of warnings are associated to bugs and that the warnings are predominately associated to refactorings. They applied Findbugs, Jlint and the static analysis component of Intellij Idea to 20 releases of 2 projects. Through the static analysis tools, they determine that they could be used to predict refactorings between releases.

2.7 Conclusion

We are able to build and run static analysis tools on over 45 thousand commits, a substantial increase over the 4.5k, 116 and 34 releases processed by previous works. Since we run static analysis on every change we are able to track when and where a warning is introduced and retroactively assign warnings to commits that did not build with our warnings recovery process. Our tool, WARNINGSGURU, allows developers to see the origin of a warning and whether the warning is new, reducing the effort involved in manually examining warnings.

Of all commits that were analysed, 58.5% of commits have a warning in one of the files which they modified. We determined that 12.8% of these commits introduced new warnings and that 6.8% of all commits had new security warnings. This reduces the effort required in the investigation of warnings that static analysis tools report. Using WARNINGSGURU, a team could prevent the introduction of new persistent warnings by proactively removing all newly introduced ones.

Future work would involve the inclusion of additional build tools as part of the WARNINGSGURU pipeline which would increase the number of project which we can support. This would also increase the number of languages which the tool can build and subsequently analyse. This would allow us to determine if this solution is applicable to other project solutions then the ones we studied. Further research would be required in the warning recovery process to improve its performance.

In addition, the use of WARNINGSGURU allows for the creation of historical build logs and artifacts which might not be preserved or available for the projects which are being analysed. These rebuilt historical artifacts, while not used in this study can be used for future work that needs historical information from the project.

Chapter 3

Statistical Models

3.1 Introduction

Statistical bug models have been used to predict the occurrences of bugs in projects. They use historically mined development information to indicate risky, *i.e.* potentially defective, files or commits (Kamei et al., 2013; Hall et al., 2012). We will call the predictors used in these models *change measures*. The two types of change measures are churn measures, such as the number of lines changed in a commit, and developer measures, such as the expertise of the developers who have modified a file. The advantage of statistical bug models is that they provide reasonable predictions of field defects in commits and files (Hall et al., 2012). The disadvantage is that the prediction is not fine-grained, *i.e.* an entire file or commit is flagged as potentially bug introducing.

We investigate the use of warnings as a change measure for statistical bug prediction models. Warnings have previously been used by Rahman et al., 2014 and Tang et al., 2015, but they were using a small number of snapshots of projects. They were also following the granularity of the snapshots which was a release that would include many changes without distinguishing when a warning was introduced specifically as WARNINGSGURU performs. We then investigate if statistical warnings models are a feasible option to identify the risk of warnings occurring in a commit. Models are generated to predict the occurrence of historical and new warnings in a commit. We apply the same methodology to warnings which are classified to be security related.

In Section 3.2 we propose an augmented statistical bug prediction model which uses the presence of warnings in a commit to determine its risk of being risky. Subsequently, Section 3.3 proposes statistical models of warning detection which identify if the commit is at risk of introducing the different types of warnings from WARNINGSGURU. In Section 3.4 we evaluate the risks that the projects have on the success of the prediction and Section 3.5 review the related work in statistical models of bug prediction. Finally, Section 3.6 summaries the conclusion of the statistical models which we propose.

3.2 Statistical Bug Models

We systematically built a series of models to assess the predictive power of warnings in predicting bug-introducing changes and compare them with COMMITGURU's change measures model. The models are logistic regressions with

the outcome measure being whether a commit introduced a bug. We determine if a commit is bug-introducing by using SZZ algorithm identifies the bug introducing commit by tracing the changes of bug fixing commits (Kim et al., 2006; Bowes et al., 2016) (*i.e.* the lines that introduced the bug). We make three models: a traditional bug model including the COMMITGURU change measures, a static analysis warnings bug model, and the combined model: COMMITGURU and warnings model. We use the percentage of deviance explained of the model to determine the quality of the model (Cataldo et al., 2009). The deviance explain is a measure of the percentage of the dataset which a model explains. We also report the odds ratio for each measures. The distribution of each measure is skewed, so non-binary variables are log_2 transformed. Table 3.1 report the model results. We describe each of the measures used in the model in the next section.

3.2.1 Measures

COMMITGURU Change Measures

The change measures in COMMITGURU have been successfully used in papers on bug prediction (Kamei et al., 2013; Kim et al., 2006). The change measures COMMITGURU model is our baseline and we compare and augment it with our static analysis warning model. The measures are aggregated at the commit level.

The change measures are divided into churn and developer measures. The churn measures include the number of subsystems and the number of directories which have modified files. It also includes number of modified files, number of lines added, lines deleted and the total line count of files before being modified. The average time between changes to a file and the number of unique changes to the file are tabulated. Entropy is a measure of the distribution of the change between modified files. The developer measures are the number of developers which modified a file, the developer's experience calculated over the entire project, the recency of experience on the modified files and a developer's experience on the modified subsystem. Additionally, the identification of a commit as bug fixing is used as a measure.

We run a Spearman correlation among the measures and keep the most parsimonious measure when the correlation is greater than 0.75. We excluded the number of modified files and entropy as they correlates at 0.94 and 0.89 respectively with the number of modified directories. We also excluded the number of changes to a file as it correlates at 0.78 with total line count before the line was modified. Due to the removal of the numbers subsystems measure, we also removed the developer's subsystem experience due to correlation of 0.74 in relation to a developer's experience.

WARNINGSGURU Warnings Measures

Unlike previous works that work with a limited number of releases (Couto et al., 2013; Rahman et al., 2014; Tang et al., 2015), we analyze over 45k commits. This allows us to trace the history of a warning. We introduce new measures, the

number of new warnings, and we are able to differentiate security warnings allowing us to determine when security warnings are introduced.

When building our warnings model we first include the number of new security warnings. We then include the number of new warnings, which is effectively the addition interaction term between the number of new security warnings + the number of new non-security warnings. We continue this process adding the the total number of security warnings and then the number of total warnings. In the model we also differentiate between the tool that found the warning, either FindBugs or JLint.

A final new measure is whether the build failed. A failed build can demonstrate problems in the code and environment and may indicate significant problems with the commit.

COMMITGURU and WARNINGSGURU Measures

We combine all the measures from the COMMITGURU and WARNINGSGURU models. We include the change measures first and the warning measures second. We also want to determine the degree of redundancy of our warnings measures compared with the COMMITGURU measures.

3.2.2 Statistical Bug Prediction Models

We used the measures in the following r models structures to generate each statistical bug prediction model. For each one of these models the dependent measure is whether the commit contains a bug, for which we are predicting. We are using the *as.factor* method which builds the prediction models in relation to one of the projects.

COMMITGURU statistical bug prediction model

> glm(Commit contains bug ~ log2(1+(Number of directories)) + log2(1+(Lines added)) + log2(1+(Lines removed)) + log2(1+(Lines before change)) + fix + log2(1+(Number of developers)) + log2(1+(Average time between changes)) + log2(1+(Developer experience)) + log2(1+(Recent developer experience)) + as.factor(Project), family=binomial(), control = list(maxit = 50))

WARNINGSGURU statistical bug prediction model

> glm(Commit contains bug ~ log2(1+(New security warnings)) + log2(1+ (Security warnings)) + log2(1+(New findbugs warnings)) + log2(1+(New jlint warnings)) + log2(1+(Findbugs warnings)) + log2(1+(Jlint warnings)) + Build failed + as.factor(Project), family=binomial(), control = list(maxit = 50))

TABLE 3.1: Bug prediction models and odds ratios

	COMMITGURU	Warnings	Combined
Num Directories	1.25 ***		1.05 *
Lines added	1.44 ***		1.39 ***
Lines removed	1.03 ***		1.02 **
Lines before change	1.15 ***		1.08 ***
FIX	1.52 ***		1.51 ***
Num of Devs on files	0.74 ***		0.82 ***
Avg time between changes	1.00		1.01
Developer Experience	1.05 ***		1.04 ***
Recent Dev. Experience	0.97 ***		0.99
new security warnings		1.02	1.08
security warnings		0.89 ***	0.93 ***
new FindBugs warnings		1.43 ***	1.10 *
new JLint warnings		1.58 ***	1.14 ***
FindBugs warnings		1.19 ***	1.07 ***
JLint warnings		1.35 ***	1.22 ***
build failed		2.00 ***	1.75 ***
Hadoop	0.56 ***	0.57 ***	0.42 ***
Ignite	0.55 ***	0.62 ***	0.48 ***
Kylin	0.84 **	1.01	0.85 **
Phoenix	0.38 ***	0.28 ***	0.26 ***
Ranger	0.42 ***	0.46 ***	0.33 ***
Tika	0.57 ***	0.98	0.65 ***
Wicket	0.82 ***	0.89 *	0.69 ***
Deviance Explained	19.5%	13.4%	22.0%
Residual	41195	44323	39928

p <0.001: ***, p <0.01: **, p <0.05: *, p <0.1: .

For each one of these models the dependent measure is whether the commit is bug introducing. Since all predictors are transformed with log_2 (except binary variables), interpretations are as follows, a twofold increase in the number of lines added increases the odds of a bug being introduced by 1.44 times, for example. The odds ratio is an evaluation of the increase or decrease in probability for a change in the size of a measure. p is the representation of the statistical significance of the measure where a smaller value implies that a measure is more statistically significant.

With the use of *as.factor* on the projects, all projects are in relation to the Commons-lang project. Depending on the model, the individual projects are either more are less likely to introduce a warning in comparison to Commons-lang and this is represented by an odds ratio.

Combined statistical bug prediction model

glm(contains bug ~ log2(1+(Number of directories)) + log2(1+(Lines added)) + log2(1+(Lines removed)) + log2(1+(Lines before change)) + fix + log2(1+(Number of developers)) + log2(1+(Average time between changes)) + log2(1+(Developer experience)) + log2(1+(Recent developer experience)) + log2(1+(New security warnings)) + log2(1+(Security warnings)) + log2(1+(New findbugs warnings)) + log2(1+(New jlint warnings)) + log2(1+(Findbugs warnings)) + log2(1+(Jlint warnings)) + Build failed + as.factor(Project), family=binomial(), control = list(maxit = 50))

The following section goes through the results of these models.

3.2.3 Change Measures Model Results

In the first column of Table 3.1 we see the results of the COMMITGURU change measure logistical regression bug model. We can see that the model explains a reasonable amount of the deviance, 19.5%. Since our predictors are skewed any non-categorical variable is transformed using log_2. We report the odds ratio for each predictor. However, since they are log_2 transformed they represent a twofold increase in the predictor. For example, a twofold increase in the number of directories touched, lines added, lines before a change makes it 1.25, 1.44, 1.15 times more likely for a bug to occur in a commit. Previous work has found that files with more churn tend to have more bugs, our findings confirm this suggesting that large changes introduce more bugs (Giger, Pinzger, and Gall, 2011).

Likewise, FIX which is the identification of a commit as being bug fixing, is binary. As a result a commit that is fixing an existing bug is 1.52 times more likely to introduce a bug. Our findings indicate that bug fixes likely touch fragile or complex code and lead to further bugs. This agrees with the findings that the number of past defects is a strong predictor of future defects (Nagappan and Ball, 2005b).

The strongest negative predictor is the number of developers who touch a file, with a twofold increase in this predictor leading to a 26% decrease in the likelihood of the commit introducing a bug.

3.2.4 Warnings Bug Model Results

In the second column of Table 3.1 we see the results of our static analysis warnings bug regression model. We can see that the model explains a smaller proportion of the deviance, 13.4%. These results provide limited support for previous smaller studies that suggesting that change measures (Rahman et al., 2014), OOP measures (Tang et al., 2015), and static analysis warnings have similar defect prediction potential. We discuss these studies in more detail in the related work section as some involve complex accounting schemes to deal with limitations in the unit of analysis and the small number of versions analysed.

The strongest predictor of bug introduction is whether the build succeeded. A build failure doubles the likelihood of a bug being introduced. A twofold

increase in the number of new warnings for a commit increases the likelihood of introducing a bug by 1.5 and 1.19 times for JLint and FindBugs respectively. Having more existing warnings in the code that is changed in a commit also increases the likelihood of introducing a bug. New security warnings were not statistically significant while the total number of security warnings actually reduced the likelihood of a bug. This final conclusion is likely related to the difficulty and rarity of actual security bugs, which as (Camilo, Meneely, and Nagappan, 2015) point out, make vulnerabilities difficult to predict statistically.

3.2.5 Combined Change and Warnings Measures Bug Model Results

In the third and final column of Table 3.1 we see the combination of the change measures and static analysis warning measures. The deviance explained is 22.0% only 2.5 percentage points higher than the change measures model. As a result, we conclude that the warnings model contributes an 11% increase in the deviance explained by change measures model. The measures are for the most part consistent in terms of direction and power with the largest drops in predictive power seen by the new warnings predictors. Indicating that these measures take into account changes in file size and complexity and the new warnings likely explain a similar phenomenon.

3.3 Statistical Warning Prediction Models

In this section we perform an empirical study to predict whether a commit introduces warnings and security warnings using the simple COMMITGURU change measures, such as lines added and developer experience. Our goal is to understand how these measures influence the introduction of warnings. For example, do developers with more experience introduce fewer security warnings? A practical outcome of this research will be to suggest which commits and files deserve further analysis with the more computationally expensive static analysis. For example, a commit that is statistically more likely to introduce a security warning could be flagged for additional analysis. As a final contribution, recent work by Camilo, Meneely, and Nagappan, 2015 showed that security vulnerabilities cannot be predicted by statistical bug models using change measures. We create models to predict security warnings that are able to flag potential vulnerabilities.

3.3.1 Description of the Warnings Models

We create four logistical regression models with the following outcomes: does the commit contain warnings, new warnings, security warnings, or new security warnings. The predictors are the same change measures used in the previous models: the number of directories, the number of lines added, the number of lines deleted, the number of lines before the change, whether the the change a fixes a bug, the number of developers that have touched the files in the commit, the average time between changes for the files in the commit, the average

developer experience for the files in the commit, and the experience of the developer who most recently changed the files under commit.

The following sections below illustrate the r models which are used. The dependent measure changes between model where we build a model that predict different type of warnings. We are using the *as.factor* method which builds the prediction models in relation to one of the projects.

Statistical warnings prediction model

glm(warnings ~ log2(1+(Number of directories)) + log2(1+(Lines added)) + log2(1+(Lines removed)) + log2(1+(Lines before change)) + fix + log2(1+(Number of developers)) + log2(1+(Average time between changes)) + log2(1+(Developer experience)) + log2(1+(Recent developer experience)) + as.factor(Project), family=binomial(), control = list(maxit = 50))

Statistical new warnings prediction model

glm(New warnings ~ log2(1+(Number of directories)) + log2(1+(Lines added)) + log2(1+(Lines removed)) + log2(1+(Lines before change)) + fix + log2(1+(Number of developers)) + log2(1+(Average time between changes)) + log2(1+(Developer experience)) + log2(1+(Recent developer experience)) + as.factor(Project), family=binomial(), control = list(maxit = 50))

Statistical security warnings prediction model

glm(Security warnings ~ log2(1+(Number of directories)) + log2(1+(Lines added)) + log2(1+(Lines removed)) + log2(1+(Lines before change)) + fix + log2(1+(Number of developers)) + log2(1+(Average time between changes)) + log2(1+(Developer experience)) + log2(1+(Recent developer experience)) + as.factor(Project), family=binomial(), control = list(maxit = 50))

Statistical new security warnings prediction model

glm(New security warnings ~ log2(1+(Number of directories)) + log2(1+(Lines added)) + log2(1+(Lines removed)) + log2(1+(Lines before change)) + fix + log2(1+(Number of developers)) + log2(1+(Average time between changes)) + log2(1+(Developer experience)) + log2(1+(Recent developer experience)) + as.factor(Project), family=binomial(), control = list(maxit = 50))

The deviance explained by the warnings models is 19.1%, 29.9%, 21.5%, and 30.3% for total warnings, new warnings, security warnings, and new security warnings respectively. The new warnings and new security warnings models are better models.

The change measures are better at predicting new warnings and new security warnings than identifying commits which have historical warnings on

TABLE 3.2: Warnings models and odds ratios

	Warnings	New Warn	Security	New Sec
Num directories	1.65 ***	1.15 ***	1.87 ***	1.07 *
Lines added	1.21 ***	1.68 ***	1.17 ***	1.64 ***
Lines deleted	1.03 ***	0.97 ***	1.05 ***	0.97 ***
Lines before change	1.24 ***	1.03 **	1.28 ***	1.04 **
FIX	1.21 ***	0.97	1.19 ***	1.09
Num Developers	0.99	1.01	0.92 ***	1.01
Time btwn changes	1.01	1.01	0.97 ***	1.00
Dev Experience	0.98 ***	1.02 **	0.98 ***	1.00
Recent Experience	0.95 ***	0.92 ***	0.92 ***	0.92 ***
Hadoop	0.47 ***	0.94	0.84 ***	1.15
Ignite	0.56 ***	0.91	1.08	1.22
Kylin	0.83 ***	1.30 **	1.08	1.62 ***
Phoenix	2.46 ***	6.27 ***	5.03 ***	14.1 ***
Ranger	0.57 ***	1.89 ***	1.32 ***	3.46 ***
Tika	0.88 *	2.02 ***	1.51 ***	3.36 ***
Wicket	0.76 ***	0.61 ***	0.65 ***	0.44 ***
Deviance Explained	19.1%	29.9%	21.5%	30.3%
Residual	50453	24628	48704	15875

p <0.001: ***, p <0.01: **, p <0.05: *, p <0.1: .

For each one of these models the dependent measures are the warnings, new warnings, security warnings and new security warnings respectively. Since all predictors are transformed with log_2 (except binary variables), interpretations are as follows: a twofold increase in the number of lines added increases the odds of a new warning by 1.68 times, for example. The odds ratio is an evaluation of the increase or decrease in probability for a change in the size of a measure. p is the representation of the statistical significance of the measure where a smaller value implies that a measure is more statistically significant. With the use of *as.factor* on the projects, all projects are in relation to the Commons-lang project. Depending on the model, the individual projects are either more are less likely to introduce a warning in comparison to Commons-lang and this is represented by an odds ratio.

their modified files. This suggests that these predictors provide more accurate representations of recent events on the project than the overall project health.

3.3.2 Warnings Model Results

The first column shows the warnings model with 19.1% of the deviance explained. The strongest predictors are the simple measures of churn. A twofold increase in the number of modified directories, the number of lines before the change, and the number of lines added increases the odds of a commit having at least one warning by 1.65, 1.24 and 1.21 times, respectively. A bug fix change also increases the odds of a warning by 1.21 times. The developer measures and time between changes have a very minor impact on the odds ratio for warnings.

3.3.3 New warnings Model Results

The second column shows the new warnings model with 29.9% of the deviance explained by the model. The strongest predictor is the number of lines added. A twofold increase in the number of lines added increases the odds of a new warning by 1.68 times. Interestingly, the more recent experience the developer has with the files in a commit the less likely he or she is to introduce a warning. A twofold increase in recent experience leads to a decrease of 8% in the number of new warnings introduced. Although statistically significant, the overall developer experience with the files under change has a very minor impact on the number of new warnings.

3.3.4 Security warnings Model Results

The third column shows the model for security warnings with 21.5% of the deviance explained by the model. The model is similar to the warnings model with the churn measures having a large impact on the number of security warnings. The amount of recent experience decreases the number of security warnings with a doubling in experience results in a reduction of 8% in the odds of having a security warning.

A predictor that is unique to the security warnings is the number of developers who have touched the files in the commit. With a twofold increase in the number of developers working on the files in a commit the number of security warnings decreases by 8%. Future work is necessary to investigate this in greater detail. A possible explanation is with more developers touching a set of files comes more attention to possible vulnerabilities, eliminating security problems before they are committed potentially through peer review.

3.3.5 New Security Warnings Model Results

The fourth column shows the model for new security warnings with the highest deviance explained of the warnings models, 30.3%. The model shows more similarity to the new warnings model than the security model. New security warnings are best predicted by the lines added and recent experience.

Our models provide adequate predictions of security warnings. Recent work by Camilo, Meneely, and Nagappan, 2015 that use similar change measures to find past security vulnerabilities, reported that statistical bug models are unable to reliably predict security vulnerabilities as reported in a common vulnerability and exposure report (MITRE Corporation, 2017). We have shown that our models have reasonable predictive power for new warnings and in particular new security warnings. It is promising that new security warnings, which are 53% more rare than general new warnings show a similar predictive ability. Our work suggests that assigning developers who have recently modified a file to review it would likely be a simple way to reduce the number of new security introduced into the system.

3.4 Threats to Validity

The techniques applied might not be effective for other types of projects. However, given that we successfully applied the benchmark to all of the projects we expect that if the benchmark functions on a test project that the combined statistical bug model would function on it as well. The same statement is also applicable to the statistical warnings model which uses the benchmark measures to build its statistical model.

3.5 Related Works

Kim et al., 2007 determines that fault are localised, whether that be there code proximity, change set or time. The measures extracted from these are then usable to predict the occurrences of bugs in a project. This is a precursor to the churn and developer measures which we use for our prediction models. Kim, Whitehead, and Zhang, 2008 uses measures obtained from the history of the project obtained from the SVC and terms from the change to predict if it is risky. Kim, Pan, and Whitehead, 2006 identifies the pattern of a bug fix in the history of the project to generate patterns which are specific to to the project. These can predict future occurrences of the same type of bug.

There is little work that examines both static analysis and statistical bug models. Couto et al., 2013 studied the bug finding effectiveness of static analysis on three projects and find that in the 280 bug fixing changes the static analysis tool FindBugs would have suggested a warning for only 33 of the changes. Furthermore, static analysis tools produce a large number of warnings with Couto et al., 2013 finding between 4 and 10 warnings per KLOC depending on the project. Factoring the multitude of warnings and the limited effectiveness of identifying commits that contain bugs, the authors find a median precision and recall for FindBugs of zero. The authors conclude that static analysis warnings do not correspond to field defects. The authors then examine a single release for 30 projects and find a moderate correlations of .56 between the number of warnings that exist in the code when the release is made and the number of bugs reported against the release. Our work tracks warnings as they are introduced to the project. The new warnings predictor is superior in predicting bugs than

the total number of warnings on a commit as used by Couto *et al.*. However, we find that the warnings model only explains 13.4% of the deviance, indicating that static analysis is a poorer predictor of bugs.

A preliminary work by Tang *et al.* Tang et al., 2015 on 2 projects and 8 revisions showed that OOP measures such as LCOM "Lack of Coupling in Methods" and McCabe complexity provide less predictive power than static analysis warnings. The predictive models were unstable across revisions suggesting that future work is necessary to replicate these finding on a larger number of project revisions.

Work by Rahman et al., 2014 concluded that "under some accounting principles, they [FindBugs, PMD, JLint] provide comparable benefits [to statistical bug models]." Unfortunately, the methodology and accounting schemes in the paper make replication and interpretation difficult as the statistical models provide predictions at the file level and the static analysis tools provide warnings at the line or code unit level requiring complex "budgeting" of warnings and statistical risk. A further limitation is that the authors process only 34 versions requiring complex git blame assignment of warnings to past revisions ignoring static analysis warnings that may have been removed between releases. Our study which aggregates at the commit level and examines over 45k commits finds that static analysis can predict bugs, but that it does less well than simple change measures.

Rahman et al., 2014 also conclude that the "performance of certain static bug-finders can be enhanced using information provided by statistical defect prediction." We find that the static analysis measures add only a small 2.5 percentage point increase in deviance explained over the change measures indicating that the computationally expensive static analysis has much redundancy with simple churn measures and will only provide limited enhancement in predictions.

Proponents of static analysis correctly argue that the warnings identified are not designed to find many classes of bugs, such as those related to user experience problems. A manual study of FindBugs by at Google Ayewah et al., 2007 found that of the 1127 warnings examined, 17% of the warnings were "impossible" meaning that they "could not be exhibited by the code". An additional 11% of the warnings were deemed to be trivial. While static analysis warnings may not increase the number of bugs found, we have developed a tool to help developers see the warnings that are present in risky commits and those that are introduced in the current commit. Future user studies are necessary to understand if limiting the number of warnings that a developer sees reduces the effort in eliminating the impossible, trivial, and false positives warnings suggested by static analysis. Nagappan and Ball, 2005a found that static analysis tool can be used to distinguish low and high quality components. They found a strong correlation between components with a high density of warnings and those with a high defect density from testing for which they created a prediction model.

Kim et al., 2011 assessed the impact of noise present in the data which is mined from software repositories. They determined that noise should not have a significant impact on the prediction models on the condition that false positives

and false negatives do no represent over 20% to 35% percent of the data set. They propose a method of identifying noise through the proximity of bugs and their similarity to identify noisy ones.

Ray et al., 2016 found that code is repeatable and predictable and that buggy code is unnatural. They perform a comparison of their statistical language model and determined that they their effectiveness was comparable to those of static analysis tools Findbugs and PMD.

Shivaji et al., 2013 worked on improving the performance of bug prediction techniques by reducing the number of measures which are computed to generate the prediction model. Goyal, Chandra, and Singh, 2015 use of statistical models to perform fault prediction models and determine that we need to find ways of reducing the number of measures of prediction which are used to reduce their complexity and understandability by their users. Canfora et al., 2013 propose a method to improve cross project defect prediction. Models trained for one project have reduced performance when applied to another project. They applied a genetic algorithm to a multi objective logistical regression model to obtain a model with improved effectiveness over standard cross project models.

Herzig et al., 2013 predicts buggy changes through change genealogy that identifies changes which have cascading issues which are not directly associated to the subsequent changes which will be buggy. The effect of a change may not be buggy, but its modifications can result in subsequent buggy commits. Giger, Pinzger, and Gall, 2011 distinguish between different types of code churns (statements, conditionals, etc...) to generate a prediction model which outperform standard code churn measure and Jiang, Tan, and Kim, 2013 built statistical prediction models personalised to target individual developers.

3.6 Conclusion

We present a tool and a series of statistical models that combine change measures and static analysis warnings to predict whether a commit will introduce a bug. We find that the model accounts for 22% of the deviance. However, there is much redundancy between the simple change measures of churn and developer experience and the computationally more expensive static analysis warnings.

We also create models of whether a change will have a warning or security warning based on the simple change measures. We are able to explain 30% of the deviance in the commits that will introduce new security warnings, which helps in the known difficult task of finding security vulnerabilities. We find that developers who have more recent experience with the files in a commit are substantially less likely to introduce new security warnings.

Future work would require the integration of the combined statistical bug model into the interface of WARNINGSGURU to provide warnings as a prediction measure for the risk of a commit. In addition, the integration of the statistical warnings prediction model into WARNINGSGURU would allow the tool to either prioritise the analysis of commits which are at risk of being warnings or excluding those that are not risky. This would reduce the required resource of WARNINGSGURU to build and analyse the commits, reducing the period until feedback can be provided for a risky commit.

Chapter 4

Usefulness of Warnings: Developer Study

4.1 Introduction

One of the purposes of WARNINGSGURU is to provide timely, relevant feedback to developers when they make commits to the software project. WARNINGSGURU can identify new warnings on commits, reducing the number of warnings which need to be investigated by distinguishing them from historical warnings. The motivations of this study is to assess whether new warnings obtained from static analysis tools are useful to developer and whether by providing these warnings in a timely matter there is an impact on their perceived usefulness of the warnings. As opposed to previous studies which either targeted a developer who was on the same team (Nanda et al., 2010) or one who was not involved in the project's development Ayewah and Pugh, 2008, we survey the developer who author the commit which introduced the warning.

We explicitly leave the definition of "useful" up to the developers and allow them to provide a comment to provide more details on utility. We find that usefulness can range from identifying a new bug where an issue is opened [1] for a path traversal attack vector to unused variables and code style issues. We provide exploratory results to the following research questions:

4.1.1 RQ 1, Usefulness & Characteristics: How many new warnings are useful and what are their characteristics?

Studies of static analysis tools assume that the warnings produced are useful to developers (Rahman et al., 2014). However, studies that examine individual warnings have found that many of them are false positives (**Ayewah:2007:ESA:1251535.1251536**; Ayewah and Pugh, 2008) which might result in developer not using static analysis tools (Johnson et al., 2013). Our goal is to understand if the new warnings identified by WARNINGSGURU are useful to developers.

We compare the utility of the warnings found by FindBugs and JLint. We also distinguish between security and non-security warnings that these tools can report.

[1] https://issues.apache.org/jira/browse/RANGER-1450

4.1.2 RQ 2, Timeliness: Does sending timely messages to developer affect the perceived usefulness of the warning?

Static analysis tools report many warnings that have existed in the system for extended time periods. To study the impact of timeliness, the messages we send to developers cover a range of times between February 1st 2017 to the 23rd of March 2017. We record the delay between the author date of the commit by the developer and the announcement of the warning to the original developer. By investigating the impact of the delay we assess if the perceived usefulness of a warning is affected in a statistically significant manner.

We define the timeliness of warnings to be the time period in days between author date of a commit and the time when we receive a response from the developer as to whether a commit is useful to them.

This chapter is divided into the methodology (Section 4.2) which includes the description of the new warnings and developers we target (Section 4.2.1) and our data collection survey (Section 4.2.2). We answer the research questions in Section 4.3 where we assess the usefulness of new warnings. Section 4.4 evaluates the risks to the study results and Section 4.5 review previous static analysis tools user studies. We conclude with Section 4.6 which reviews the results of the study.

4.2 Methodology

We performed a user study of developers who introduced new warnings as part of a commit. Using WARNINGSGURU, we are able to identify commits which had new warnings. The purpose of this study is to determine if the warnings are useful to the developers who introduced the warning. The developers were contacted by email and were provided with the opportunity to indicate if the warning was deemed useful to them. We purposely did not provide the developers with a definition of useful and allow them to leave a comment to define how the warning was useful. The study includes commits that were committed to the selected projects between the period of the 1st of February 2017 to 23rd of March 2017.

4.2.1 Data

As part of WARNINGSGURU, we have demonstrated that we are capable of identifying the warnings which are attributable to the change that a developer as performed. 14.4% of commits introduce new warnings which represents 173 commits that introduces warnings for 80 unique developers as part of our previous dataset of projects in the WARNINGSGURU pipeline. We therefore increase the number of projects that WARNINGSGURU analyses from 8 to 33.

For this study we analysed the following Apache projects: Accumulo, Apex core, Apex malhar, Asterixdb, Beam, Brooklyn-server, Calcite, Canyenne, Cloudstack, Commons-lang, Commons-net, Commons-text, Crunch, Curator, Falcon, Hadoop, HBase, Ignite, Tamaya-Extension, Knox, Kylin, ManifoldCF, Oozie, OpenNLP, Phoenix, Ranger, Sentry, Storm, Tika, Tinkerpop, Twill, Wicket and

TABLE 4.1: Number of commits and developers between the 1st of February 2017 and the 23rd of March 2017

Commits	with new Warnings	Developers	with new Warnings
3572	435	468	183

Number of commits and unique developers for the 37 analysed projects

Zeppelin. These are all projects which meet the requirements for WARNINGS-GURU (See Section 2.3). These projects have a total of 435 commits which introduce at least one new warning from a total of 183 unique developers based on the email address used in the commit during the user study period (Table 4.1).

Overall the commits with new warnings are spread between the remaining projects with a high of 53 in Beam and a low of 1 Cloudstack, Crunch, Falcon and Sentry. When we consider that Beam has the most commits during the time period (775 commits), it is expected that it would have the greatest number of commits with new warnings. These values are reflected in the number of developers who introduced new warnings in the commits of a project (Table 4.2). The projects which have the most commits also have the most developers who are active during the study period (Ignite, Hadoop, HBase and Beam).

The number of developers in Table 4.3 will not add up to the values in Table 4.1 since some developers contribute to more than one of the projects which we analysed. We consider each developer to be unique based on their email address. However where two developers have the same name and their email are similar, we consider them to be the same person for the survey portion of the study.

4.2.2 Survey

The following criteria were followed for the survey. To be eligible, a commit needs to introduce one or more new warnings as detected using WARNINGS-GURU. The commit needs to have been committed between the 1st of February 2017 and the 23rd of March 2017. For developers who have previously received a survey request, there is a minimum of 7 days between survey requests. This prevents the developers from being overwhelmed by multiple survey requests. Developers who have multiple email addresses, based on a manual verification of name and email, are considered to be one developer and we follow the standard delay between submission of survey requests.

The survey is composed of two parts: an introduction email and the survey page which it links to. We dynamically generated the contents of both of these components to reflect the commit and the new warnings which it included. Each one of these parts are manually validated to ensure that the links and page are functional before contacting the developer.

Survey Email

The solicitation is an email that is sent out to the developer who authored the commit. This email is sent out to the email address that is associated with the commit and its content is personalised with the name of the author. If the name

TABLE 4.2: Commits with new warnings in selected Apache projects during user study period

Project	Commits	With warning	With new warnings
accumulo	58	21	11
apex-core	28	12	4
apex-malhar	42	20	9
asterixdb	116	65	33
beam	775	202	53
brooklyn-server	155	84	18
calcite	68	48	14
cayenne	92	55	15
cloudstack	108	15	1
commons-lang	74	47	2
commons-net	50	36	4
commons-text	46	11	2
crunch	6	3	1
curator	19	4	2
falcon	14	11	1
hadoop	273	195	44
hbase	174	134	40
ignite	249	110	45
knox	47	24	5
kylin	176	117	23
manifoldcf	56	18	4
oozie	34	19	3
opennlp	43	18	8
phoenix	54	45	19
ranger	135	82	19
sentry	12	8	1
storm	96	24	6
tamaya-extensions	49	23	8
tika	67	46	13
tinkerpop	239	70	6
twill	16	10	2
wicket	95	45	10
zeppelin	95	47	9

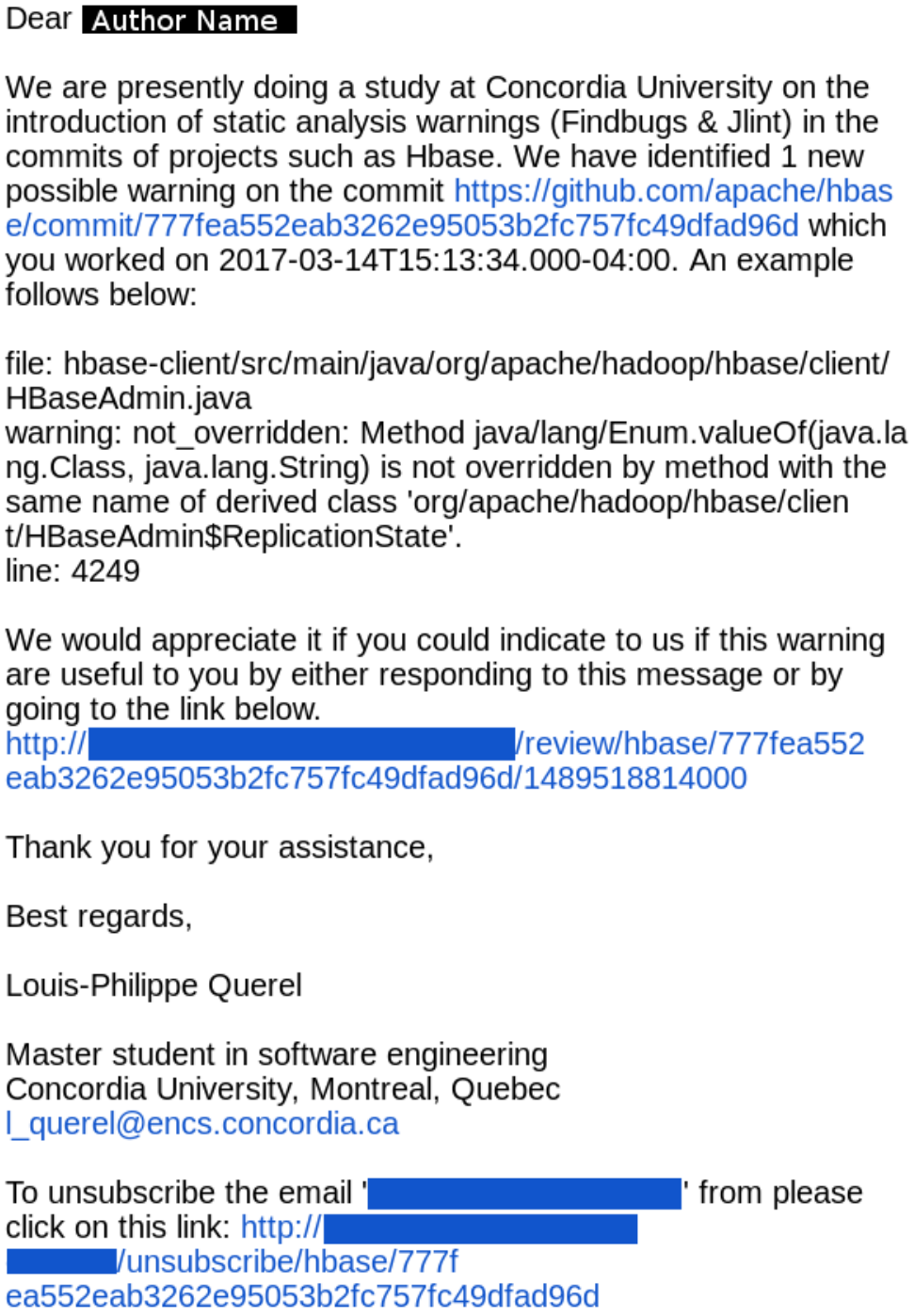
There are a total of 3572 commits in the user study period, 435 of which have one or more new warning as described in Table 4.1.

TABLE 4.3: Developers by projects during the user study period

Project	Developers	Introduced New Warning
accumulo	7	4
apex-core	13	4
apex-malhar	18	4
asterixdb	13	8
beam	54	16
brooklyn-server	15	4
calcite	20	8
cayenne	4	1
cloudstack	18	1
commons-lang	12	2
commons-net	1	1
commons-text	4	2
crunch	4	1
curator	7	2
falcon	5	1
hadoop	53	22
hbase	39	20
ignite	41	23
knox	3	3
kylin	16	9
manifoldcf	3	1
oozie	9	3
opennlp	8	3
phoenix	16	11
ranger	16	10
sentry	2	1
storm	19	4
tamaya-extensions	3	2
tika	4	2
tinkerpop	13	2
twill	6	2
wicket	7	3
zeppelin	31	6

465 distinct developers authored a commit during the study period. 183 of these developers authored a commit which introduced one or more new warnings as described in Table 4.1. As some developers participated in more then one project the sum of the table will not equal these values.

FIGURE 4.1: Sample survey email


Dear Author Name

We are presently doing a study at Concordia University on the introduction of static analysis warnings (Findbugs & Jlint) in the commits of projects such as Hbase. We have identified 1 new possible warning on the commit https://github.com/apache/hbase/commit/777fea552eab3262e95053b2fc757fc49dfad96d which you worked on 2017-03-14T15:13:34.000-04:00. An example follows below:

file: hbase-client/src/main/java/org/apache/hadoop/hbase/client/HBaseAdmin.java
warning: not_overridden: Method java/lang/Enum.valueOf(java.lang.Class, java.lang.String) is not overridden by method with the same name of derived class 'org/apache/hadoop/hbase/client/HBaseAdmin$ReplicationState'.
line: 4249

We would appreciate it if you could indicate to us if this warning are useful to you by either responding to this message or by going to the link below.
http://[redacted]/review/hbase/777fea552eab3262e95053b2fc757fc49dfad96d/1489518814000

Thank you for your assistance,

Best regards,

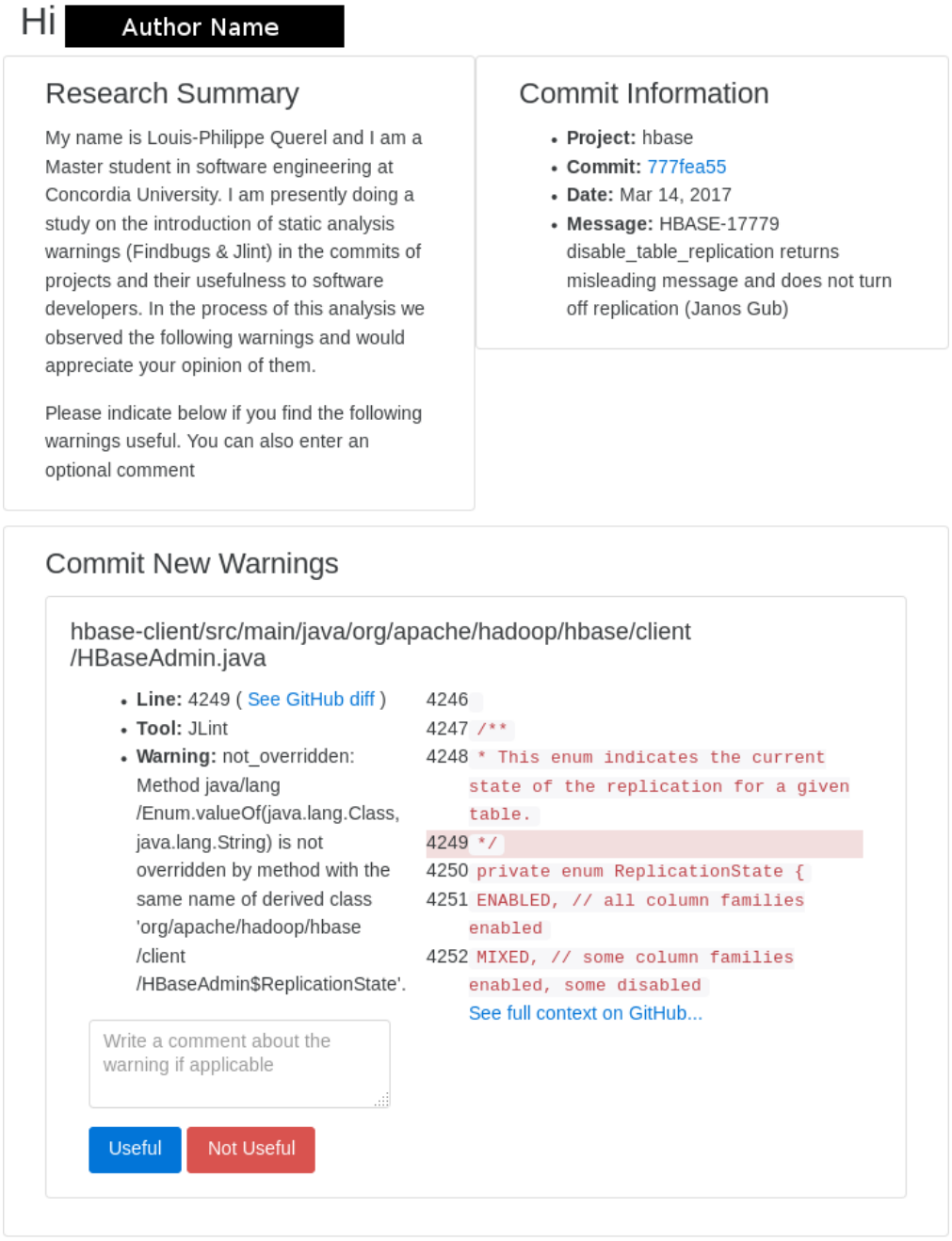
Louis-Philippe Querel

Master student in software engineering
Concordia University, Montreal, Quebec
l_querel@encs.concordia.ca

To unsubscribe the email '[redacted]' from please click on this link: http://[redacted]/unsubscribe/hbase/777fea552eab3262e95053b2fc757fc49dfad96d

Sample survey email that includes one warning and request to participate for the survey. This email is personalised for each commit with the developer's name and one of the new warnings introduced in the commit.

FIGURE 4.2: Sample survey page



Sample survey page which present commit and new warnings that have been identified. The developer can specify if each warning is useful for them and provide feedback. This page is personalised for each commit to include the new warnings that were identified in the commit.

of the developer does not appear to be valid, we instead obtain the name from the project's list of contributors. The email informs the developer that we are undertaking a study on static analysis warnings and that our analysis of the

commit has identified one or more new warnings associated to the commit. One of the warnings is presented as a sample which includes the file name, line number and warnings details. The developer is than encouraged to provide their feedback on the warnings by either responding to the email or going to the link of the unique survey page. Figure 4.1 illustrates an example of the email which is sent out.

We provide a method for the developer to unsubscribe from the survey by clicking on a link which is part of the email. When we receive a response from a developer that requests to be unsubscribed we do not send them any future survey requests.

Survey Page

The survey page is where the developer is presented with their commit and the new warnings which we have identify with WARNINGSGURU. We provide the developer with description of the study and details of the commit such as the commit identifier hash, commit message and a link to the complete commit hosted on GitHub. We then present each new warning individually with the warning description, name of the tool which identified it, the line number and file location in the project repository. To assist the developer, we provide the context of the warning by presenting the lines of code and its surrounding code as part of the warning. We ask if the developer finds the warnings useful. We however do not provide the developers with a definition of what useful is and leave it to their understanding. A text field is provided with each warning for the developer to optionally provide comments. An example of a survey page is presented in Figure 4.2.

4.3 Results

Applying the methodology we sent 214 survey requests to validate the usefulness of warnings to a total of 179 developers. We received responses through the survey page which we provided to collect the data and through direct email responses. Where a direct email response was submitted we would validate the response with the developer to ensure that they either categorise the warning as either useful or not useful. The results were then consolidated by manually entering them on the survey page with the comments which the developer provided in their email.

Of these survey requests, we have obtained responses for 15.9% of them. Within the time frame of the study certain developers were sent more than one request and we received multiple responses from them. A total of 17.9% percent of developers who were contacted responded to one or more of the requests which we sent to them. Table 4.4 provides the response rate.

TABLE 4.4: Number of commits for which an email was sent out and the number of developers involved

Commit Surveys Sent	Response %	Developers	Response %
214	15.9%	179	17.9%

TABLE 4.5: Useful Warnings by Static Analysis tools

Tool	Warnings	% Useful	% Not Useful
JLint	47	21.3	78.7
Findbugs	34	52.9	47.1
Total	81	34.6	65.4

4.3.1 RQ 1, Usefulness & Characteristics: How many new warnings are useful and what are their characteristics?

The total from Table 4.5 shows that only 34.6% of warnings are useful. This could be a sign of false positives which are contained in the new warnings which we identified (Ayewah et al., 2007). By comparing by the different the static analysis tools (FindBugs or JLint) and their state of warnings as security warnings we can observe different outcomes to the results.

Warnings per Static Analysis Tool

By differentiating by the tool which generate the warning we observe that warnings from Jlint represent a greater number of responses as opposed to FindBugs warnings: 58% to 42% respectively based on 47 and 34 warning responses. We observe that there is a difference between the number of warnings which are deemed to be useful by the tool as well. 52.9% of Findbugs warnings are useful as opposed to 21.3% for Jlint warnings (Table 4.5). We applied the Fisher Exact test to this distribution and its p-value was less then 0.05 which implies that the comparison of the two datasets are statistically significant. The difference between the number of responses for Jlint is overwhelmingly skewed to warnings not being useful which would explain the 34.6% in Table 4.5

Warnings per Security Classification

We investigated whether security warnings was a statistical significant characteristic of the warnings. WARNINGSGURU categorises warnings as to whether the warning is security-related. We use this distinction to determine if security warnings are deemed more useful to developers compared to non-security warning in relation to the static analysis tool which identified it. Table 4.6 includes both of these comparisons. We analysed the security warnings and non-security warnings individually.

Security Warnings Warnings which are categorised as security-related represented 38% of the responses which we obtained. By distinguishing them between their respective tools we observe that they are not perceived as useful

TABLE 4.6: Useful Warnings by Security Classification

Tool	Security	Warnings	% Useful	% Not Useful
Jlint	True	10	20.0%	80.0%
	False	37	21.6%	78.4%
Findbugs	True	21	42.9%	57.1%
	False	13	69.2%	30.8%

by the developers. Only 42.9% and 20.0% of Findbugs and Jlint security warning respectively are considered to be useful. However, given that the difference in usefulness between Findbugs and Jlint security warnings is not statistically significant we cannot extract a conclusion from the comparison.

Non Security Warnings The comparison between non security warnings of Jlint and Findbugs was determined to be statistically significant. By observing the warnings which have been categories as being non-security-related, Jlint warnings have a similar useful rate of 21.6% to all Jlint warnings. 69.2% of Findbugs warnings which are not classified as security-related are perceived as useful. This illustrates that non security warnings from Findbugs are perceived as useful to developers.

Comparing between the security classification of warnings cannot be done due to the security warnings being not statistically significant, but if the results from the security warnings are validated to be true, that would imply that non-security warnings are more useful in the case of Findbugs. This would require the re-evaluation of the assumed pertinence of security warnings.

4.3.2 RQ 2, Timeliness: Does sending timely messages to developer affect the perceived usefulness of the warning?

We have determined that certain categories of warnings are useful to developers, such as warnings from Findbugs in most cases. The responses which we received were for warnings that were introduced throughout the study period. Some of the survey requests were sent out for commits which had occurred over 6 weeks prior to the request and as a result contained warnings that were as old as the commit. While the survey requests which were sent out all contained different warnings, the selection is random in that we did not target specific types of warnings in the surveys. We only presented the warnings which were new to the commit.

We want to determine if the timeliness of the notification to the developer has an impact on the perceived usefulness of the warning. We calculate the time delta in days between the introduction of a new warning by a developer and the time which we receive a response to our survey request. The time delta to the response is then associated to the warning and the usefulness state which the developer reported.

Based on the survey result, we determined that responses to useful warnings occurred in a median delta of 11.5 days between their introduction in the commit

and the developer response. For warnings which were indicated to be non-useful by developers, there was a median delta of 23 days. We obtain a p-value of 0.018 from performing a Wilcoxon test on the set of useful and not-useful warnings. Given that this value is below the value of 0.05, we can say that the two datasets are statistically significant. Figure 4.3 illustrates the skew that warnings which have a smaller time delta are more useful. However, a longer delta does not mean that a warning will not be useful. We observed useful warnings at 43 days, which is also the greatest value for not useful warnings as well.

These results would indicate that developers tend to find newer warnings to be more useful. It is therefore important to provide developer with timely feedback on the warnings that their commits introduce in the software project. This would allow them to find the warnings more useful.

A preliminary hypothesis is that the developers forget the context of the commit as more time passes, which results in the developers identifying the warnings as not being useful. This would require additional research to assess if this could be an analog in software projects for the forgetting curve which is the study how people remember and recall information (Averell and Heathcote, 2011). Complex code changes and complex warnings would require additional context to be able to understand them and possibly evaluate the usefulness of warnings associated to it.

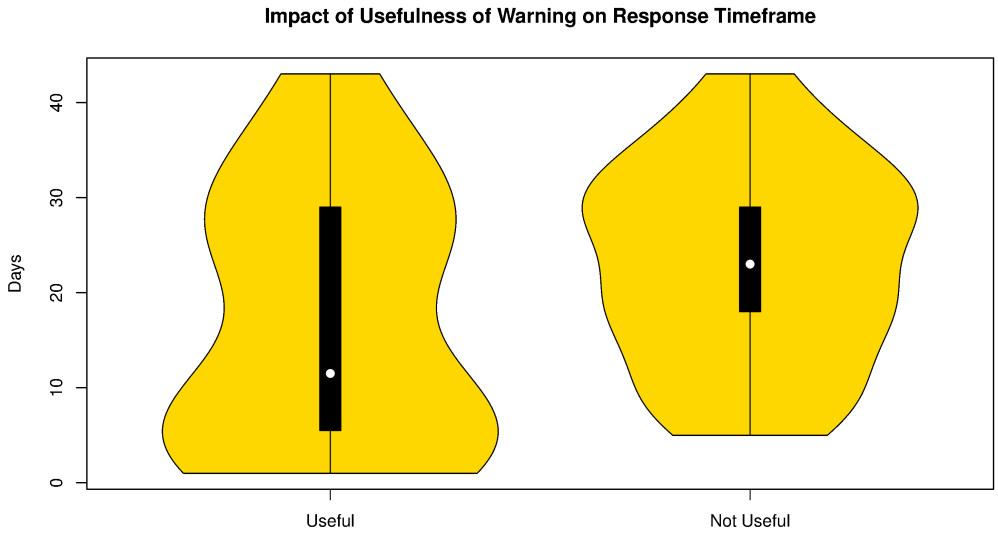
4.3.3 Responses From Developers

As part of the study we were also collecting comments from the developers. The comments by the developers were split between the usefulness of the warnings. On one survey, one warnings might be useful while another might not be for a developer. Below, we review some of the main topics which we obtained from the developers who participated.

There were multiple situations where the warnings were associated to the test cases of a project. The developer in such case would inform that it was a test case and would explain why it was not an issue. There was at least one instance where the developer was impressed that the static analysis tool found a specific type of valid warning in the tests of the project. Some other situations dealt with defective code that had voluntarily been added in an improper way to validate a test case. It was also perceived that there was a smaller importance overall by the developers for these type of warnings. An improvement would be to filter warnings contained in test files to present fewer of them to the developers.

Three developers responded that they would open an issue on the project regarding the warning which we presented to them. An extension to this were cases that the warnings identified parts for which the developer or project were already aware that an issue was present. The specific warning had in one case already been addressed by a subsequent commit. These statements indicate that some of the warnings from static analysis tools are indeed useful at identifying issues in a software project. Some projects were already using static analysis and in a few cases WARNINGSGURU reported additional warnings which led them

FIGURE 4.3: Impact of the notification timeframe on the usefulness of warnings



These violin plot represent the distribution of warnings based on their plot. The dot represents the median of the dataset and the box plot and below and above the medians are the representations of the second and third quartiles respectively.
The notification of warnings as an impact on the warning perceived usefulness where developers perceive more warnings to be useful where the notification delay between the commit time and survey response is smaller.

to inquire about the tools and configurations which we are using to perform our static code analysis.

Developers also questioned the solutions which the warnings proposed. The developers instead identified their current solution to be either more appropriate or easier to understand. In this case it would be recommended that more descriptive messages should be provided to provide the developer with the context of why an approach might be better. Some of the responses also seem to imply that some developers did not understand the concepts of synchronisation in parallel computing or that the static analysis tools were wrong in its warnings.

There is also at least one case where the static analysis tools identified impossible warnings by indicating that there was an issue on a line for which there was no code. We could address this issue by adding a sanity verification which would validate that the warning is indeed associated to a functional line of code. However this raises the question of how frequently a static analysis tool might fail to correctly identify a warning to the right line of code.

Finally, in the developers explanations and evaluations of different warnings there were also many expressions of gratitude. While this might not be scientific, it did give the impression that even if the warnings were deemed not to be useful, the results that we were presented to them were being appreciated by

some of the developers.

4.4 Threats to Validity

It is possible that our responses are not representative of developers of open source software. However we targeted 37 projects to allow us to have the greatest number of possible developers to complete the survey. The responses represented provided results for both tools as two distinct distributions.

Some projects have static analysis tools configured as part of their project build, but as we have observed we still obtained warnings for the projects where these were configured. Some of these warnings were also useful to the developers which would indicate that they are either not running the same tools or their configurations is filtering useful warnings. It is therefore still possible to obtain useful warnings from these projects, but there might be an over representation of non-useful warnings. However our evaluation of timeliness of warnings imply that this would not be the only reason for a greater number of non-useful warnings.

4.5 Related Works

Previous user studies of static analyser have been performed. Ayewah and Pugh, 2008 performed a user study of Findbugs and another static analysis tool to determine if the warnings should be fixed, had minimal impact or was not a bug. They completed a questionnaire to assess the warnings. Lewis et al., 2013 completed a bug prediction user study where they determined that there was optimism to predictions of bugs, but that current tools lacked the required maturity for day to day frequent use.

Johnson et al., 2013 interviewed 20 developers and evaluated that false positives in the results were a recurring issue. They also reported that the presentation of the results could have an impact on the use of the warnings, whether this be the layout of the warnings or the messages and descriptions attributed to them. There is also a need to ensure that these tools can work and be shared within the workflow of a team so that an entire team may use them as oppose to an individual member.

Nanda et al., 2010 determined that the results from different static analysis should be merged and filtered when presented to developers. They developed a tool that runs static analysis tools on snapshots of a project and which can report which warnings originate from the current change in addition to filtering the results. As part of their research they performed user studies which obtain positive results from developers.

4.6 Conclusion

We completed a user study on the usefulness of new warnings identified by WARNINGSGURU by surveying 179 developers who introduced new warnings in 214 commits. We obtained a 15.9% response rate. By differentiating by the

static tools, we observed that 52.9% of new warnings identified by Findbugs are useful and only 21.3% of Jlint warnings are useful. While static code analysis tools might not be able to identify bugs such as the placement of a button in an interface, the warnings which it identifies can be useful to developers

WARNINGSGURU also classifies the warnings based on their risk of being security-related. A study of this classification was inconclusive for security warnings, but for non security warnings, 69.2% of Findbugs warnings were deemed to be useful by the developers who responded. Like warnings overall, the usefulness of non security warning is partially attributable to the static analysis tool which identifies the warnings

By investigating the time delta until a developer is advised of new warnings in a commit, we conclude that warnings are deemed more useful if they are reviewed within a median of 11.5 days after their introduction. If it takes a median of 23 or more days the warning will be more likely be not to useful. We propose that developer should receive the new warnings attributed to their work promptly to ensure that they are perceived as useful. Future work should look into the forgetting curve to evaluate if the context of the commit as an impact on the warnings usefulness.

Based on the comments that we obtained from developers, the warnings from static analysis tools can be useful to them, but additional filtering would be required to exclude lower priority and impossible warnings from being presented to them. Furthermore, the warnings should have additional documentation associated to them so that the context of the warning can be understood more easily.

Chapter 5

Conclusion

We built WARNINGSGURU which builds historical commits and run static analysis tools on their build artifacts. In Chapter 2 this is applied on the over 45,000 commits of 8 Apache projects. WARNINGSGURU extracts warnings in 58.5% of commits and determines that 12.8% of all commits introduced new warnings and that 6.8% of all commits had new security warnings. In subsequent runs of the WARNINGSGURU, it is expanded to analyse 37 Apache projects for a total of over 55,000 commits which are analysed by the pipeline by the conclusion of the user study. WARNINGSGURU is capable of managing multiple concurrent Maven based projects and identifies the new warnings which are introduced in each commit.

We then demonstrate two approaches that uses the warnings data extracted by WARNINGSGURU. We add the warnings measures to statistical models of bug prediction and determine that the new measures are effective, but only at a 2.5% point increase over the standard benchmark. This implies that warning are usable to predict the occurrence of bugs in software projects, but their impact is minimal. We then apply the benchmark measures for statistical bug model to a statistical warnings prediction which as a deviance explained of 30.3% for new security warnings. It is therefore possible to predict if a commit might be at risk of introducing a warning in the project.

The second approach is the user study which evaluates if new warnings extracted by WARNINGSGURU are perceived to be useful by the developers who introduced them. We sent 214 survey request and had a responses rate of 15.9%. From these responses we assessed that the usefulness of the warning is partially dependent on the tool which generated it. 52.9% of new Findbugs warnings were deemed useful as opposed to only 21.3% of new Jlint warnings identified by WARNINGSGURU. We also evaluate the timeliness of informing the developer of a new warning by evaluating the impact of time on the usefulness of warnings. We concluded that developers who are informed of the warning in a median of 10.5 are more likely to indicate it as useful as oppose to a median of 23 day for warnings that were not useful.

Future work would entail improvements to WARNINGSGURU which would allow it to build and analyse a greater number of commits and projects. Filtering the warnings of WARNINGSGURU based on criteria obtained through the user study such as test and impossible cases can reduce the number of non-useful warnings that are presented to the developer.

WARNINGSGURU can also be improved by the result of our statistical models

and user study. By introducing statistical warnings models into WARNINGSGURU, it would prioritise the analysis of commits which have a greater chance of introducing new warnings and new security warning. This would improve the effectiveness by further reducing the number of execution of static analysis tools which it needs to perform and providing a more timely response to identify commits with new warnings. Also, by adding the addition of warnings measures to COMMITGURU's prediction model would improve the risk assessment of commits.

These improvements would in turn serve to improve the models and subsequent user studies. Finally, based on the user study we would need to study the impact that forgetfulness curve has on the developers' perceived usefulness of warnings.

Bibliography

Averell, Lee and Andrew Heathcote (2011). "The form of the forgetting curve and the fate of memories". In: vol. 55. 1. Elsevier, pp. 25–35.

Ayewah, Nathaniel and William Pugh (2008). "A Report on a Survey and Study of Static Analysis Users". In: *Proceedings of the 2008 Workshop on Defects in Large Software Systems*. DEFECTS '08. Seattle, Washington: ACM, pp. 1–5. ISBN: 978-1-60558-051-7. DOI: 10.1145/1390817.1390819. URL: http://doi.acm.org/10.1145/1390817.1390819.

Ayewah, Nathaniel et al. (2007). "Evaluating Static Analysis Defect Warnings on Production Software". In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE '07. San Diego, California, USA: ACM, pp. 1–8. ISBN: 978-1-59593-595-3. DOI: 10.1145/1251535.1251536. URL: http://doi.acm.org/10.1145/1251535.1251536.

Beller, M. et al. (2016). "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1, pp. 470–481. DOI: 10.1109/SANER.2016.105.

Bevan, Jennifer et al. (2005). "Facilitating Software Evolution Research with Kenyon". In: vol. 30. 5. New York, NY, USA: ACM, pp. 177–186. DOI: 10.1145/1095430.1081736. URL: http://doi.acm.org/10.1145/1095430.1081736.

Bowes, David et al. (2016). "Mutation-aware Fault Prediction". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: ACM, pp. 330–341. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931039. URL: http://doi.acm.org/10.1145/2931037.2931039.

Camilo, Felivel, Andrew Meneely, and Meiyappan Nagappan (2015). "Do bugs foreshadow vulnerabilities? a study of the chromium project". In: *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE, pp. 269–279.

Canfora, Gerardo et al. (2013). "Multi-objective Cross-Project Defect Prediction". In: *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. ICST '13. Washington, DC, USA: IEEE Computer Society, pp. 252–261. ISBN: 978-0-7695-4968-2. DOI: 10.1109/ICST.2013.38. URL: http://dx.doi.org/10.1109/ICST.2013.38.

Cataldo, Marcelo et al. (2009). "Software dependencies, work dependencies, and their impact on failures". In: vol. 35. 6. IEEE, pp. 864–878.

Couto, Cesar et al. (2013). "Static correspondence and correlation between field defects and warnings reported by a bug finding tool". In: vol. 21. 2, pp. 241–

257. DOI: 10.1007/s11219-011-9172-5. URL: http://dx.doi.org/10.1007/s11219-011-9172-5.

Cyrille Artho (2017). *Jlint*. http://jlint.sourceforge.net/.

Giger, Emanuel, Martin Pinzger, and Harald C. Gall (2011). "Comparing Fine-grained Source Code Changes and Code Churn for Bug Prediction". In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. MSR '11. Honolulu, HI, USA: ACM, pp. 83–92. ISBN: 978-1-4503-0574-7. DOI: 10.1145/1985441.1985456. URL: http://doi.acm.org/10.1145/1985441.1985456.

Goyal, Rinkaj, Pravin Chandra, and Yogesh Singh (2015). "Comparison of M5'Model Tree with MLR in the development of fault prediction models involving interaction between metrics". In: *New Trends in Networking, Computing, E-learning, Systems Sciences, and Engineering*. Springer, pp. 149–155.

Hall, T. et al. (2012). "A Systematic Literature Review on Fault Prediction Performance in Software Engineering". In: vol. 38. 6, pp. 1276–1304. DOI: 10.1109/TSE.2011.103.

Herzig, Kim et al. (2013). "Predicting defects using change genealogies". In: *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, pp. 118–127.

Jiang, Tian, Lin Tan, and Sunghun Kim (2013). "Personalized defect prediction". In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, pp. 279–289.

Johnson, Brittany et al. (2013). "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, pp. 672–681. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486877.

Kamei, Yasutaka et al. (2013). "A Large-Scale Empirical Study of Just-in-Time Quality Assurance". In: vol. 39. 6. Piscataway, NJ, USA: IEEE Press, pp. 757–773. DOI: 10.1109/TSE.2012.70. URL: http://dx.doi.org/10.1109/TSE.2012.70.

KDM Analytics (2016). *Blade Tool Output Integration Framework (TOIF)*. http://www.kdmanalytics.com/toif/.

Kim, Sunghun and Michael D. Ernst (2007). "Which Warnings Should I Fix First?" In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE '07. Dubrovnik, Croatia: ACM, pp. 45–54. ISBN: 978-1-59593-811-4. DOI: 10.1145/1287624.1287633. URL: http://doi.acm.org/10.1145/1287624.1287633.

Kim, Sunghun, Kai Pan, and E. E. James Whitehead Jr. (2006). "Memories of Bug Fixes". In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '06/FSE-14. Portland, Oregon, USA: ACM, pp. 35–45. ISBN: 1-59593-468-5. DOI: 10.1145/1181775.1181781. URL: http://doi.acm.org/10.1145/1181775.1181781.

Kim, Sunghun, E. James Whitehead Jr., and Yi Zhang (2008). "Classifying Software Changes: Clean or Buggy?" In: vol. 34. 2. Piscataway, NJ, USA: IEEE

Press, pp. 181–196. DOI: 10.1109/TSE.2007.70773. URL: http://dx.doi.org/10.1109/TSE.2007.70773.

Kim, Sunghun et al. (2006). "Automatic Identification of Bug-Introducing Changes". In: *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. ASE '06. Washington, DC, USA: IEEE Computer Society, pp. 81–90. ISBN: 0-7695-2579-2. DOI: 10.1109/ASE.2006.23. URL: http://dx.doi.org/10.1109/ASE.2006.23.

Kim, Sunghun et al. (2007). "Predicting Faults from Cached History". In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE '07. Washington, DC, USA: IEEE Computer Society, pp. 489–498. ISBN: 0-7695-2828-7. DOI: 10.1109/ICSE.2007.66. URL: http://dx.doi.org/10.1109/ICSE.2007.66.

Kim, Sunghun et al. (2011). "Dealing with noise in defect prediction". In: *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, pp. 481–490.

Lewis, Chris et al. (2013). "Does Bug Prediction Support Human Developers? Findings from a Google Case Study". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, pp. 372–381. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486838.

MITRE Corporation (2016). *Common Weakness Enumeration (CWE)*. https://cwe.mitre.org/.

— (2017). *Common Vulnerabilities and Exposures (CVE)*. https://cve.mitre.org/.

Nagappan, Nachiappan and Thomas Ball (2005a). "Static Analysis Tools As Early Indicators of Pre-release Defect Density". In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: ACM, pp. 580–586. ISBN: 1-58113-963-2. DOI: 10.1145/1062455.1062558. URL: http://doi.acm.org/10.1145/1062455.1062558.

— (2005b). "Use of Relative Code Churn Measures to Predict System Defect Density". In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: ACM, pp. 284–292. ISBN: 1-58113-963-2. DOI: 10.1145/1062455.1062514. URL: http://doi.acm.org/10.1145/1062455.1062514.

Nanda, Mangala Gowri et al. (2010). "Making Defect-finding Tools Work for You". In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ICSE '10. Cape Town, South Africa: ACM, pp. 99–108. ISBN: 978-1-60558-719-6. DOI: 10.1145/1810295.1810310. URL: http://doi.acm.org/10.1145/1810295.1810310.

Philippe Arteau (2017). *Find Security Bugs - FindBugs Plugin*. https://find-sec-bugs.github.io/.

Rahman, Foyzur et al. (2014). "Comparing Static Bug Finders and Statistical Prediction". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, pp. 424–434. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568269. URL: http://doi.acm.org/10.1145/2568225.2568269.

Ray, Baishakhi et al. (2016). "On the "Naturalness" of Buggy Code". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16.

Austin, Texas: ACM, pp. 428–439. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884848. URL: http://doi.acm.org/10.1145/2884781.2884848.

Rosen, Christoffer, Ben Grawi, and Emad Shihab (2015). "Commit Guru: Analytics and Risk Prediction of Software Commits". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, pp. 966–969. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2803183. URL: http://doi.acm.org/10.1145/2786805.2803183.

Shivaji, Shivkumar et al. (2013). "Reducing features to improve code change-based bug prediction". In: vol. 39. 4. IEEE, pp. 552–569.

Spacco, Jaime, David Hovemeyer, and William Pugh (2006). "Tracking Defect Warnings Across Versions". In: *Proceedings of the 2006 International Workshop on Mining Software Repositories*. MSR '06. Shanghai, China: ACM, pp. 133–136. ISBN: 1-59593-397-2. DOI: 10.1145/1137983.1138014. URL: http://doi.acm.org/10.1145/1137983.1138014.

Tang, Hao et al. (2015). "Enhancing Defect Prediction with Static Defect Analysis". In: *Proceedings of the 7th Asia-Pacific Symposium on Internetware*. Internetware '15. Wuhan, China: ACM, pp. 43–51. ISBN: 978-1-4503-3641-3. DOI: 10.1145/2875913.2875922. URL: http://doi.acm.org/10.1145/2875913.2875922.

The Apache Software Foundation (2016). *Maven - POM Reference*. https://maven.apache.org/pom.html.

University of Maryland (2017). *FindBugs*. http://findbugs.sourceforge.net/.

Wedyan, Fadi, Dalal Alrmuny, and James M. Bieman (2009). "The Effectiveness of Automated Static Analysis Tools for Fault Detection and Refactoring Prediction". In: *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*. ICST '09. Washington, DC, USA: IEEE Computer Society, pp. 141–150. ISBN: 978-0-7695-3601-9. DOI: 10.1109/ICST.2009.21. URL: http://dx.doi.org/10.1109/ICST.2009.21.

Wong, W. Eric et al. (2016). "A Survey on Software Fault Localization". In: vol. 42. 8. Piscataway, NJ, USA: IEEE Press, pp. 707–740. DOI: 10.1109/TSE.2016.2521368. URL: http://dx.doi.org/10.1109/TSE.2016.2521368.

Yi, Kwangkeun et al. (2007). "An Empirical Study on Classification Methods for Alarms from a Bug-finding Static C Analyzer". In: vol. 102. 2-3. Amsterdam, The Netherlands, The Netherlands: Elsevier North-Holland, Inc., pp. 118–123. DOI: 10.1016/j.ipl.2006.11.004. URL: http://dx.doi.org/10.1016/j.ipl.2006.11.004.

Zheng, Jiang et al. (2006). "On the Value of Static Analysis for Fault Detection in Software". In: vol. 32. 4. Piscataway, NJ, USA: IEEE Press, pp. 240–253. DOI: 10.1109/TSE.2006.38. URL: http://dx.doi.org/10.1109/TSE.2006.38.

Appendix A

Pipeline Architecture