Efficient Reliability Verification Testing in Open Source Software using Priority Prediction

by

Patrick Pape

A dissertation proposal submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 10, 2015

Keywords: fault injection, data-flow analysis, reliability, open source, test case prioritization and minimization, static analysis, software metrics, machine learning, logistic regression, classification algorithms

Approved by

John A. Hamilton Jr., Chair, Alumni Professor of Computer Science and Software Engineering
David Umphress, Associate Professor of Computer Science and Software Engineering
Xiao Qin, Associate Professor of Computer Science and Software Engineering
Jeff Overbey, Assistant Professor of Computer Science and Software Engineering

Abstract

Open source software is becoming a strong alternative to private development for a wide market of applications. There is a stigma against using open source software in the private sector because of licensing restrictions and the desire for closed source proprietary technologies. The idea of using open source software is often a more risky, though cheaper, alternative to private development. This dissertation presents a novel procedure for efficient reliability verification testing for open source software. The procedure utilizes both static and dynamic analysis to prioritize methods and variables within those methods for testing. Test case prioritization and minimization is used to create a more efficient process for identifying key areas in the code for error handling. Fault injection testing is done relative to the intended use cases of the system that will integrate the open source software. The results of the analysis and testing are used to calculate a metric of relative importance that is used to determine the highest priority locations for error detection and correction mechanisms. The metric data collected for the modules, functions and variables is used in conjunction with a two-step predictive model to reduce the total time needed to test the software to locate these critical variables in the system. The first stage uses binary classification and metric data about the modules and functions to identify which functions contain the most critical variables with respect to reliability against data faults. The second stage uses metric data about the variables in these functions and logistic regression to predict the relative importance of the variables. This testing procedure allows for greater flexibility in choice of open source software to integrate into an existing system through efficiently identifying key areas in the code that are the most susceptible to data faults that lead to system failure.

Acknowledgments

I would like to thank my committee members for their patience and guidance throughout the process of completing this dissertation. Particular thanks goes out to Dr. Hamilton who was instrumental in helping me to guide the path of research for this dissertation and also broadening my view as to the possible implications of the research and potential areas for future work. My thanks go out to my colleagues from Auburn and MSU who let me bounce my ideas off of them and gave valuable feedback and suggestions for potential research leads.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This research is based on previous work presented as a methodology for increasing the dependability of open source software components [1]. This dissertation takes that work as a foundation and focuses on using test case prioritization and minimization techniques to make the process more efficient and accurate. This will be done in three main components: static analysis, dynamic analysis, and fault prediction. The master's thesis referenced above succeeded in showing that the process for identifying key variables in a system for error handling was successful in selecting the variables with the highest chance of failure and that it was possible for future research to make the process more efficient.

The first stage of the research deals with refining the static analysis potion of the process that utilizes metrics such as lines of code, code slicing and memory analysis. To create a more accurate process and to allow for a greater prediction rate of high failure rate variables, the new testing framework incorporates several additional metrics to the existing process. The refined process includes code complexity and accounts for the likelihood of bit faults corrupting data relative to the modules, functions and variables stored in memory. In the previous work, it was assumed that all functions held the same chance for a single transient data fault to occur, when in reality that is not the case. Combining the memory analysis with code complexity and other static analysis metrics aids in making the prioritization process more efficient.

The second stage of the improved testing framework refines the dynamic analysis portion of the process that identifies relative modules in the open source software and in predicting potential propagation of faults throughout the system. The refined process leverages the code coverage metric to ensure that the maximum number of potential paths are examined for each part of the code and allows the testing framework to establish metrics that are relative to the usage of the system by accounting for which code does or does not get executed in the test cases. Combining the code coverage metrics with test case prioritization and minimization techniques augments the current dynamic analysis techniques in removing unnecessary tests and narrowing down potential placement locations for error handling. In addition to augmenting the placement locations for error handling, the expanded dynamic analysis testing allows for a more accurate prediction of variable priority when used in conjunction with machine learning.

The third stage of the improved testing framework works along with the current fault-injection testing techniques created previously to utilize various metrics found through static and dynamic analysis to increase the accuracy of the prediction of relative importance of the variables to increase the efficiency of the testing phases, resulting in fewer, but more accurate, total tests for the software. This is done by utilizing a two stage predictive model that consists of machine learning algorithms trained on the metrics collected by the framework. This predictive model flags functions in each module of the software system as high or low priority and then ranks the relative variables in the high priority modules according to their relative importance to the reliability of the system. The goal when integrating this predictive model into the testing framework is to reduce the number of direct interactions taken with the system to collect data, meaning less fault injection and intrusive testing methods that take up important testing time.

1.1 Problem Statement

By utilizing relative use cases and test case prioritization and minimization techniques, it is possible to reduce some of the risk that is inherent to adopting open source software for use in a real-life applications. There are several issues with integrating open source software into a system or adopting open innovation policies into a firm that is focused on commercial products. This dissertation focuses on the idea that open source software may be a cheaper alternative to private development, but that it is also a much riskier alternative. Using open source software as either the foundation or to add functionality to a commercial product is gaining momentum. The research objective is reducing risk by creating an efficient process for completing reliability verification testing of open source software through predicting the priority of variables in the system.

The software is analyzed with both static and dynamic techniques and is put through a series of fault-injection tests to identify the priority modules, functions and variables in the software. These techniques allow for the minimization of test cases by focusing only on the use cases relative to how the software will be used by the system that is integrating it. Anything not directly related to the use of the system is culled, and the test cases are prioritized according to the likelihood that the variables will lead to a system failure. This metric and fault injection data is used to teach machine learning algorithms to predict whether a function in a software system is of high priority, meaning that it is part of the critical data-flow through the software, and then to rank the variables in each of those high priority functions in terms of their relative importance to the reliable execution of the software in the presence of transient data faults.

This likelihood of system failure is a major concern for trying to incorporate open source software into a product with mission critical functions. An example would be using an open source alternative for a mission control system in a UAV. The mission control system is responsible for the control of the UAV and for communication between the UAV and the ground station or controller. If the software is unreliable, then the risk is too high to warrant the cost savings for using an open source alternative. So, this dissertation looks at how the software will be used and identifies the key variables in the system. Then, by measuring the chance for failure and the impact of the failure across the system, variables can be singled out for error handling mechanisms to prevent this occurrence. This research focuses primarily on coding defects and data corruption through simulated transient faults in memory, but it is possible to expand the focus of the research to security and software vulnerabilities. An example of a transient data fault with respect to the UAV control system, would be extreme temperature changes that effect the data stored in the local memory of the UAV. This temperature changes causes the data stored in memory to fluctuate during the execution of the system. This dissertation seeks to identify and prioritize the variables in the system to show which variables, when affected by fluctuating data, would cause a failure in the system operation.

The current architecture from [1] that forms the foundation of the research presented in this dissertation has several areas that needed improvement to keep up with current research trends. There were also assumptions made for the sake of initial testing and validation of the methods for identifying the critical path through the software component and placement of error handling mechanisms. This dissertation takes the next step to replace the assumptions with real measures of memory analysis and other factors to more accurately predict the likelihood of failures in variables throughout the code.

1.2 Benefits

The popularity of open source software and open innovation within the private and public sector rises each year. This research produces results from a process capable of taking an open source software component in post-release and efficiently increasing the reliability of the components for use in a pre-existing system. This allows the use of open source software in more commercially oriented markets and is an early step in furthering the idea of open source software as a strong alternative to consumer-off-the-shelf or in-house development.

The process described in this dissertation is novel in the way that it combines different metrics of static and dynamic analysis with fault-injection testing and machine learning in order to identify high priority variables for placing error handling mechanisms. The return to the research community is the data from the hybrid analysis and testing process completed on open source software cases. The results of the machine learning algorithms for the predictive model provides feedback in a process that leverages the prediction of high priority variables for test case priority and minimization.

This testing framework has uses outside of the open source community. Assuming access to the source code in question, this framework is useful for helping to manage the increasing time and cost of completing full testing suites on software in the late stages of development or during regression testing from a software patch. Though there is room to expand functionality of the tool, it could be used along with statistical or historical data analysis tools to create a more efficient means of prioritizing test cases. The tool could also be expanded to consider security aspects of software and focus on the detection of vulnerabilities or weaknesses in the code.

Chapter 2

Literature Review

In this portion of the dissertation, related publications in topics that apply to the research discussed previously in the introduction will be detailed. The review focuses on several points: open source software, test case prioritization and minimization and software reliability. The major goal of the dissertation research is to refine the procedure used in the referenced master's thesis to prioritize variables in open source software using a mix of static and dynamic analysis and machine learning. The key research emphasis is on aspects of developing, maintaining and integrating open source software into real-world systems and making the prioritization process more accurate and efficient.

2.1 Open Source Software

Open source software usage is rising in the way that organizations are developing and commercializing software. This section discusses the related research on adopting and commercializing open source software, in addition to current research on detecting bugs and quality issues specific to open source software. The goal of this dissertation research is not to determine the use of open source software in the real-world or to explore the economic and business aspects of commercializing it. An underlying premise is that open source software is becoming an increasingly popular solution to reducing development time and costs and therefore we review current research to verify this premise.

2.1.1 Adoption

In [2] Ayala, Cruzes, Hauge and Conradi detail five facts about adopting open source software by various types of organizations, both private and public. Of particular interest here is one of the ways that open source software is adopted by private section organizations. Open source products are being integrated into other products or systems in an effort to increase software reuse through reducing licensing, upgrade costs and development time. An example is integrating some open source components into a system, extending some functionality of the open source component or wrapping it. Utilizing open source software has some hidden costs: training in the product, licensing costs with adding functionality, support, expenses dealing with commercial strategies, backward compatibility and selecting the right open source software for the system.

A review of the lessons learned from adopting open source software is shown by Fitzgerald, Kesan, Russo, Shaikh and Succi in [3]. They make several assertions based on their results, but the most important relative to this research is that open source software adoption is not successful in all environments. There are certain conditions that need to be met and inherent risks when dealing with open source. This means that when choosing which open source software would be a good fit as a test case in the dissertation research, it is important to note the likely use cases and the type of project that would incorporate the software.

An integrated model for adopting open source software is tested and evaluated by Gwebu and Wang in [4]. The model utilizes several key aspects before deciding whether to adopt open source software: identification, personal innovativeness in technology, perceived usefulness, and perceived ease of use. For this research, the perceived usefulness and ease of use are most likely the aspects that are affected. The research seeks to make the procedure to prioritize the placement of error handling mechanisms more accurate and efficient. Doing this will reinforce the decision to switch to open source if the error handling is sufficient.

A major consideration when integrating open source software into a pre-existing system is the consequence of changing the code or adding new functionality. Adam Reed discusses various economic aspects of dealing with open source software [5], but more importantly notes that improvements to the open source software are likely to be made once an organization integrates the software. This means considering the implications of the GNU public license if the software will then be re-distributed as part of a product.

The issue of viewing technology as a proprietary element and what that means for open source software is discussed in [6] by Spinellis and Giannikas. Open source is such an appealing option because it is free, the source code is available, and it can be modified to meet the organizations needs. But, depending on how the software is to be used, it is not really an option for use in purely proprietary software products. The study lead to the conclusion that companies are more likely to profit when focusing on their own core competencies and operations using products readily available on the market. But, one caveat was that for organizations of all sizes, open source software was seeing increasing use in the realm of web servers and depending on the organization, open source software could be utilized in other areas.

2.1.2 Commercialization

The commercialization aspect of dealing with open source software was discussed somewhat in the previous research, but the next two papers deal specifically with that challenge. The open source development model's place in commercial companies is explored by Frank Hecker in [7]. The topic of licensing surfaces again in the article when considering taking the development of the commercial software in an organization through an open source development method. The products begin as closed source, consumer-off-the-shelf type software and eventually end up as open source software to ease the maintenance and featurization costs that come post-release. The various licensing options are discussed and the developer is encouraged to select one that best fits their needs. For example, GNU places tighter constraints on releasing code commercially without contributing changes to the code to the developer community, whereas licenses like the Berkeley Software Distribution and the Mozilla Public License are more lax when it comes to proprietary options.

Commercial open source software products are those that are privately developed based on publicly available source code. These types of products are ideal for the research in this dissertation. By efficiently and accurately placing error handling, the open source option becomes even more ideal for this type of development. Kumar, Gordon, and Srinivasan [8] develop a two-sided model of competition between commercial open source software firms to address the issue of dealing with competition, since portions of the source code are now freely available to other firms. The model shows that mandatory feature sharing between firms can raise profits and that the availability of open source contributions improves the quality of products more than just private development.

2.1.3 Open Innovation

A paper by Bonaccorsi and Rossi [9] on why open source can succeed discusses three problems that occurred with the increase in open source software usage: motivation, coordination and diffusion. The problems lead to a series of key questions about why open source was developed, if not for profit, and how effective can the coordination of strangers around the globe be on a project? The focus of the paper is more on the business aspect of open source software, but is worth noting because of the high consideration open source has been getting over the last decade. The study showed that more research was needed to determine the impact open source software would have on the market, but it was noted that an increase in the usability of open source software will result in increased use, as the user is the key focus in commercial software.

The idea of a firm that focuses on open source technology is not a brand new concept, Red Hat is the largest contributor to Linux. Colombo, Piva and Ross-Lamastra discuss the concept of open innovation and diversification in the industry in [10]. The paper explores the open innovation paradigm with respect to small and medium sized enterprises. The study showed that the diversification was negatively associated with the size of the enterprise and positively associated with the number of open source projects that the enterprise contributes to. The relationship between innovation and diversification of the industry is further explored by investigating the extent to which the enterprises collaborate with the open innovation partners. That is, the relationship between the open innovation enterprises and the development communities that contribute to the same open source projects.

One of the key aspects of the research is the focus on open source software, but the methods used could also be of use to any software where the source code is available. The concept is that the software is in post-release when you attempt to integrate, so standard methods for testing and placing error handling will not suffice. A paper by Li, Kivett, Zhan, Jeon, Nagappan, Murphy, and Ko [11] discusses the differences between pre- and post-release versions of software. The study observed application crash and hang, system crash and usage information from millions of Windows users. The goal was to examine how usage characteristics affected field quality, pre- and post-release and to report the data back to Windows. The four major usage characteristics for examining the field quality of Windows were: number of applications executed, pre-installation, language locales, and 32 versus 64-bit. The study showed that there were prediction errors for the anticipated defects of about thirty-six and eighty-nine percent for two different releases of Windows 7. The results show that there is a large margin of error going from beta tests to actual field testing post-release.

Midha and Palvia explore the factors that make open source software successful in [12]. The success of open source software was measured over a period of time to reflect the evolutionary nature of open source. Though the concept of success in software development can be subjective, the study indicated success through the amount of user interest and developer contribution to a project. It was shown that the level of user interest may increase the developer contribution. A more popular project was more likely to see updates in the developer community by those wanting to be part of a popular project. Other factors were considered when observing changes in user interest and developer contribution, such as: license choice, size of user and developer base, language translations, responsibility assignment, complexity, modularity, and release. It was found that projects that coordinated responsibility, focused community attention on key parts of the software and allowed for the maximum amount of users led to the most successful projects.

The last paper considering open innovation looks at the challenges to a firm investing in open source software. West and Gallagher [13] identify three major speed bumps to dealing with open source software: ”finding creative ways to exploit internal innovation, incorporating external innovation into internal development, and motivating outsiders to supply an ongoing stream of external innovators.” [13] The study observed firms dealing with open source software to support their innovation strategies. Put more simply, the issue that was being investigated was how do you make a profit from something that is inherently free to the customer? A common answer was to use the open source software as a foundation to start some other intellectual property or to use the input of the developer community in addition to internal developers to create something better. The balance is that when the firm takes the lead and uses external developers, they incur a higher coordination cost, but less risk. If the open source community is the lead on a project, more risk is involved for a reduced cost.

2.1.4 Bugs

With one of the key focuses of the research being on the open source aspect of the code to be tested, it is important to get an idea of the types of bugs that are more commonly found in open source software than in privately developed software. Cotroneo, Grottke, Natella, Pietrantuono, and Trivedi explore various fault triggers based on different types of bugs in [14]. The paper analyzes types of bugs in four large open source software projects throughout their life-cycle. The research in this paper is useful for understanding the bug characteristics leading to a system failure and identifying the best way to deal with various bugs in different phases of the project. Of particular interest are the methods for dealing with bugs that are most likely to be found later in the development stages. These methods included: model checking, stress testing, code reviews, and error detection and correction mechanisms.

Further characterization of bugs found in open source software is seen in [15], by Li, Tan, Wang, Lu, Zhou, and Zhai. The paper discusses how the nature of bugs has changed in open source software. The study determined that memory-related bugs have decreased with the rise of effective error detection tools, with the exception of certain memory-related bugs such as NULL pointer de-referencing. Semantic bugs were also found to be a dominant cause of failures in open source software. The data in this paper is useful for investigating the error handling in the framework, as it is more focused on checking for errors that are likely in open source.

Security is a major consideration in many open source software projects and the impact on security when planning bug fixes is discussed in [16], by Saleem, Yu, and Nuseibeh. The research investigates the process for planning bug fixes and whether security-related bugs need to be fixed differently than other bugs. The bugs are classified as either security or non-security related and then the frequency of the occurrence of key security terms is measured for each bug description. It was shown that it was possible to plan a tool-supported release for Samba, given that the open security issues at the time of release were known. Also, the study gives a good example of how to use an open source software project's documentation when creating a release plan.

A example of analyzing bugs found in a particular type of open source software system is shown in [17], by Yin, Caesar, and Zhou. The paper explores the bugs found in open source router software. It uses bug databases for two routers utilizing open source implementations, Quagga and XORP, and Cisco IOS security advisories and the Linux IP stack. Of key importance in this paper, is the data showing that a mere 4 percent of the code contain more than twenty-five percent of the bugs. This meant that lines of code was not necessarily a useful metric for considering the number of bugs in the software. A methodology is presented for classifying the bugs found in the router software using both static analysis and manual classification. The research gives some insight into the considerations made when focusing on the detection of bugs in an open source software system.

2.1.5 Quality

The next two papers take a look at producing quality open source software in the current market. The first article, by Mark Aberdour [18], draws some interesting conclusions from actual empirical data on open source quality. The difference between quality development in open and closed source are considered, including management, testing process, and the use of an open community of developers who can freely contribute to a code-base. Higher quality is found in projects that utilize professional resources, structured testing, automated regression testing, and user testing. The existence of a sustainable community is also a factor in the project's success. The community is found to provide some key objectivity to the testing process and when augmented with some formal testing techniques, the quality should increase.

Crowston, Wei, Howison, and Wiggins discuss more empirical research on free, libre and open source software, FLOSS, in [19]. The paper presents a quantitative summary of articles for analyzing the inputs, processes, emergent states, and outputs of such software. The paper is a useful source of information about different types of open source software and what type of considerations could be made for creating a more efficient analysis process in the research. The research found in the paper is still in an early stage, but there is a clear relationship between social interaction and software development that needs to be explored further to form a more theoretically sound understanding of the development process.

2.1.6 Validation and Verification

The idea of validation and verification testing is not new research, but the concept of validation and verification of open source software is an area that has grown larger in the last decade. The idea of using empirical validation metrics in order to predict faults in open source software is discussed in [20], by Gyimothy, Ferenc, and Siket. The paper shows calculations of object-oriented metrics to illustrate fault detection of source code. The values were checked against a bug database, Bugzilla, using regressions and machine learning methods for validation. A large number of metrics are discussed in the paper: weighted methods per class, depth of inheritance tree, response for a class, number of children, coupling between object classes, coupling between classes and lines of code. Many of these metrics will be considered in the dissertation research, so the results of the study here are useful. The most important observations to the research were that coupling between classes was found to be the best metric for fault prediction and that lines of code was a strong initial indicator of faults, but falls behind multivariate models for more fine-grained analysis.

A distributed approach to validation and verification is considered in [21], by Yuan and Gygi. The paper presents an approach for validation and verification of the Electronic Structure TEST software, run on a network of distributed servers. The case study presented in the paper is a useful reference for using a software framework for validation and verification of networking software. More important to the research, is the description of the validation and verification techniques used to analyze the ESTEST plug-ins and the challenges of implementing the open source framework.

2.2 Test Case Prioritization and Minimization

This section will review the second key aspect of the research, the process of efficiently identifying key variables for the placement of error handling mechanisms. The dissertation research is based off an existing process found in [1]. The static analysis, dynamic analysis and fault prediction methods discussed here are used to assist in creating a more accurate and efficient process.

2.2.1 Static Analysis

The related research for the static analysis to be done in the research includes the metrics of code complexity, prioritization, and others. The first paper covers validation of complexity of code changes and predicting bugs in open source software. Chaturvedi, Bedi, Misra, and Singh [22] apply linear regression to predict the release time of a project and measure performance using various statistics. The complexity of the code and the bugs previously fixed in the project are used to predict the next release problem that will occur. It was found that given that open source software does not follow a specific set of rules for releases, the method discussed in the paper is a useful way to determine the schedule for new releases. The data presented is useful for determining when to release based on bug fixes in the code and the complexity.

Dit, Revelle, Gethers, and Poshyvanyk [23] give a survey of papers dealing with the location of features in source code. The locations are used to identify initial places in the code that implement functionality in a software system. This information would be useful for choosing a starting place for static analysis to find potential propagation of faults, given a set of use cases. The taxonomy presented highlights the need for more benchmarks and comparisons in the field of feature location. The findings are useful for locating further research in the area if the concept of feature location looks like a promising means of increasing the efficiency of the static analysis in the research.

Through evaluation of code complexity, code churn and developer activity metrics, Shin, Meneely, and Williams locate software vulnerabilities in [24]. The study attempts to see if metrics obtained from source code and development history can be used to predict the location of vulnerable code. Of particular interest to the research is the process for predicting the location of vulnerabilities and how the metrics mentioned above can be used to do so. The assumptions in the study were that vulnerable files had a larger complexity and code churn than those without vulnerabilities. It was found that the code churn and developer activity metrics were more accurate in predicting vulnerabilities than code complexity. The results indicate that pursuing historical data about bugs found in open source software could increase the accuracy of the prediction process.

Sinha, Dey, Amin and Badkoobehi [25] detail the process of calculating software complexity using a variety of metrics. The software complexity metric is indicative of the testability, maintainability, readability and the comprehensibility of a program. Previously, code complexity was measured by lines of code, but that alone is not a good measure of how hard to understand the software is. The paper presents a quantitative method for measuring software complexity using code abstractions in state-chart diagrams to demonstrate. The code abstractions are modeled in the state-chart diagrams and then analyzed with the following criteria: number of execution paths, number of edges, nested loops, object references, data-structures, external files, GUI, interfaces with other systems, lines of code, login, dealing with secure user inputs. Each criteria has a scoring system in place that results in a total complexity index for the diagram.

Thomas, Hemmati, Hassan, and Blostein [26] use topic models and static analysis to prioritize test cases. Test suites can grow too large to maintain with many potentially unnecessary tests. The paper discusses a method for development teams to prioritize their test suite so that the highest number of distinct faults can be located in the fewest number of test cases. In this case, the test case prioritization is done without access to the source code in a black-box environment. A topic model is created using text analysis and linguistic data to approximate the functionality of each test case. Test cases with the highest chance to detecting a failure are prioritized over those with less functionality. Compared to other test case prioritization techniques, requiring execution behavior, the black-box static testing in the paper is capable of standing on its own. But, the methods discussed in the paper would be best served complementing a prioritization technique with code access.

The idea of predicting defective code to allocate limited resources effectively is discussed in [27], by Zhang, Zhang, and Gu. The paper measures code complexity through three key metrics: lines of code, McCabe's Cyclomatic Complexity and Halstead's Volume. The software components are classified as either defective or non-defective aided by the measurement of code complexity. The study reinforces the concept that static analysis can be used to measure component quality and predict defective modules. The method of calculating code complexity and the data from the validation testing of the predictions are important to the research, as they can be used in the static analysis to create a more accurate process for prioritizing test cases. The data from the study shows that it is possible to build effective defect prediction models based on the code complexity measures used.

2.2.2 Dynamic Analysis

A large number of following papers utilize the KLEE LLVM, described in [28] by Cadar, Dunbar and Engler. KLEE is " a symbolic execution tool that is capable of generating tests that achieve high-coverage on a diverse set of complex and environmentally-intensive programs". [28] KLEE was thoroughly tested against all 89 stand-alone programs in the GNU Coreutils utility suite, which are commonly considered to be the most heavily tested open source programs available. KLEE works by focusing on two key aspects of dealing with code: complexity and environmental dependencies. In an initial test against MINIX's tr tool, KLEE generated 37 tests that traversed all of the executable statements in tr. KLEE has two major goals: to hit every line of executable code in a program and to detect if any input exists that could cause an error at each potentially dangerous operation. The KLEE symbolic execution testing suite will be thoroughly experimented with in an attempt to leverage the capabilities of the system with the techniques in [1] to create a more efficient method for placing error handling mechanisms.

The dynamic analysis portion of the research will be based on ensuring maximum code coverage and using symbolic execution with regression testing. The first three papers come from Cadar, Sen and other contributors using the KLEE LLVM for symbolic execution and software testing. Cadar, Godefroid, Khurshid, Pasareanu, Sen, Tillmann, and Visser [29] utilize the program analysis technique known as symbolic execution along with dynamic test generation and generalized symbolic execution and report their results. There are challenges to utilizing symbolic execution with the current state of software engineering. The scalability of symbolic execution when dealing with an exponentially increasing number of paths in the code is a major issue. Also, removing redundant paths and using a heuristic search method are needed to fully implement the technique. It is noted that parallelization could be helpful in alleviating some of these problems. The concepts discussed in this paper are useful to the research because the process will use relative test cases, cutting down on the number of potential paths through the code.

Cadar and Sen [30] discuss the changes between symbolic execution as introduced in the 1970s and modern symbolic execution. Current research has revisited the concept of symbolic execution with advances in algorithms, constraint satisfiability and scalable dynamic approaches. Symbolic execution is once again an effective technique for generating high-coverage test suites and finding errors in complex software systems. Symbolic execution is used to visit as many different program paths as possible in the shortest amount of time. This makes it an ideal tool for creating a more efficient testing phase. The major strength of symbolic execution is that there is no need to provide concrete inputs to trigger specific bugs, because symbolic execution can locate bugs on any path in a system. Though more simple errors, such as buffer overflows, are possible to locate, symbolic execution can also be used to locate higher-level issues dealing with complex program assertions.

Kaushik, Salehie, Tahvildari, Li, and Moore [31] utilize dynamic prioritization techniques to create a more efficient means of performing regression testing. Because regression testing has a high cost in both resource and process management, prioritizing the test suites is especially important. The challenges of re-ordering test cases based on dynamic prioritization is addressed in this paper. To make the regression testing more efficient, in-process events and up-to-date test suites are used to measure the usefulness of the tests and determine which tests would be best. With most prioritization techniques, it is assumed that there is a fixed pool of test cases with available coverage metrics for all test cases. This provides three types of challenges: environment and resource, metric and in-situ. The paper focuses on the in-situ changes, meaning those based on events that occur within the process or situation in the test cases.

Kumar, Mishra, and Misra propose a novel approach for minimizing and prioritizing test suites in [32]. They utilize a path selection strategy for regression testing that enables the tester to execute test cases in the order that most increases the effectiveness of finding faults. Also, a genetic algorithm based technique is used to select a subset of the prioritized test cases. Of particular interest is the path selection for regression testing. The data from the study shows that the performance of the methods presented result in a greater efficiency than random testing for faults.

In an effort to incorporate less traveled paths in symbolic execution increasing the code coverage, Li, Su, Wang, and Li [33] utilize the length-n subpath program spectra. This method systematically approximates the full path information for the paths explored through symbolic execution. Frequency distributions are explored for length-n subpaths to identify the less traveled parts of the code. Finding these less traveled parts of the code increases the code coverage and the detection of errors. The KLEE LLVM testing suite is used for the symbolic execution and the data is retrieved from testing done on the GNU Coreutils programs. It is determined that the proposed method yields higher code coverage test cases than traditional symbolic execution methods.

Marinescu and Candea [34] explore automated techniques for identifying errors in application recovery code using fault injection. A framework is presented for writing precise triggers that deliver the desired faults in the form of error return codes. The reaction to these error return codes is monitored to evaluate the effectiveness of the recovery code. The framework described in the paper uses static analysis of library binaries to determine the error codes that can be used in the fault injection. The binary is analyzed for potentially vulnerable locations for the faults to be located. The applicable portion of this research comes from the combination of fault injection testing with efficient testing of recovery code.

Marinescu and Cadar [35] propose an automated technique based on symbolic execution that attempts to increase the quality of software patches. The set of test cases is generated automatically to cover most of the statements in the software. The coverage is measured by the number of lines of code covered by the test suite. The KLEE symbolic execution engine forms the base for the technique and is augmented by the code that combines the concrete inputs with the symbolic program exploration, ZESTI. The proposed technique successfully created inputs to cover all accessible path code in the test case studies using the lighttpd web server.

Another paper by Marinescu and Cadar [36] furthers the concept of high-coverage testing for software patches. In this paper, the issue of automation of testing code patches is addressed. The technique from [35] is combined with novel heuristics with static and dynamic analysis allowing for the specific code for the patch to be tested. The technique has taken the form of a tool called KATCH, which has been tested against nineteen patches written over a period of six years for an assortment of open source projects. There are several important aspects to consider from this research. The first is that even with these new developments achieving a high code coverage percentage is still difficult, only thirty-one percent coverage was obtained here. Next, automation is key in achieving high code coverage. The data showed anywhere from a twenty-one to a thirty-eight percent increase in coverage using automatic techniques.

Mirzaaghaei, Pastore, and Pezze [37] propose a framework, *TestCareAssistant (TCA)* that implements algorithms for the evolution of test suites written in Java. This article focuses primarily on the issue of updating test suites automatically. The framework is tested on five different open source projects and the results are presented. The results shows that TCA is capable of generating test cases that cover software behaviors that are not seen originally in the test cases. The article also identifies eight algorithms to aid in the automation process depending on the particular scenario found in the software. These scenarios include: adding or removing a parameter from the method signature, changing the type of the return parameter, extending a class by overloading and overwriting a method.

The challenge of testing a synchronous reactive system is addressed by Schrammel, Melham, and Kroening in [38]. The challenge comes from long input sequences that are often needed to drive the reactive system to the desired feature. The issue is particularly noticeable when dealing with on-target testing. This is the case where the system is tested in a real-life application environment where the time to reset the test and start again is high. This research is particularly important because the dissertation research takes advantage of real-life applications of the software to be tested and uses it to focus test cases. [38] presents an approach for discovering a test case chain for the software that uses a single execution that covers all the test goals in one execution, while minimizing the overall execution time. The reported experimental results come from a prototype tool for generating C code using SIMULINK models. This is then compared to state-of-the-art test suite generators. The implementation and results indicate a similarity to the KLEE LLVM.

Siddiqui and Khurshid [39] present a novel approach to scaling symbolic execution that implements a program analysis technique for a systematic exploration of execution paths. The test case inputs are automatically generated for each of the bounded execution paths. The key aspect of the research is that the state of the analysis can be compactly represented using a range, beginning and end, based on test case input for a specific symbolic execution run. Combining the KLEE symbolic execution engine with this ranged symbolic execution technique alleviates some of the issues with attempting to scale symbolic execution. The novelty of the approach is that the symbolic execution is directed by the test inputs, and not that test inputs are found through symbolic execution. This allows the symbolic execution to be broken down into sub-problems using the range of test inputs and increasing the scalability.

A novel approach for increasing the efficiency of symbolic execution given recent advances in constraint solving technology and computing power is discussed in [40], by Siddiqui and Khurshid. In this paper, symbolic execution is broken down into stages in order to compute abstract symbolic inputs. This differs from standard symbolic execution, which searches the code for paths all at once. The abstract symbolic inputs are shared across the different methods in the software to systematically test each input. This approach is evaluated and compared against the KLEE engine, resulting in cost savings for systematic testing using symbolic execution. The method described isolates and removes a large amount of the overhead in symbolic execution by creating a library of abstract symbolic tests that are used to enable symbolic test suites to recreate symbolic states as needed for testing.

The paper by Siddiqui and Khurshid [41] based on [40] presents further analysis of the potential scaling aspects of the symbolic execution done through staged execution. Of importance in this paper is the test suite size reduction which is capable through black-box techniques that use pre-conditions and class invariants. There are two key issues to doing the reduction with the black-box techniques: high number of tests, redundant tests, and bounds dependent method they are testing. This means that not all of the possible inter-leavings and overlappings will be covered and many test cases will go to waste. It is important to note that given access to the source code, these black-box techniques are less effective than other methods.

The last paper considered here is by Yoo and Harman [42] and is a survey of minimization, selection and prioritization techniques for regression testing. Test suites are known to grow to un-manageable sizes as software evolves, making it too costly to attempt to run the entire test suite. This paper is a collection of recent studies for dealing with the costly process of running huge test suites on software. The papers are separated into the three categories mentioned in the title. Test case minimization refers to eliminating redundant test cases in order to lessen the total number of tests. Test case selection identifies test cases that are particularly relevant based on recent changes to the code. Test case prioritization orders the test cases in a way that maximizes fault detection in the code. This paper serves as a strong resource for locating additional leads for future research.

2.2.3 Fault Prediction

Utilizing test case prioritization in order to predict or locate key artifacts in the source more efficiently is the focus of the last section of the literature review. The surveyed research will vary by method, but the overall focus of the research is on the testing process for finding and dealing with faults. The first paper is by Benameur, Evans and Elder [43] and deals with software development using type-unsafe languages, such as C and C++. It is critical to ensure that code is free of security vulnerabilities and bugs. A novel architecture, MINESTRONE, is presented that utilizes static analysis, dynamic confinement, and code diversification to locate and contain vulnerabilities in software. The idea came from the fact that even with security protocols in place throughout the development phase, MINESTRONE is highly unlikely to catch all potential problems. The data showed that by combining the various elements mentioned above, the detection rate of vulnerabilities was improved nearly thirty-five percent over Valgrind, a state-of-the-art tool. Of particular interest here is the use of KLEE in the test suite generation for detecting the bugs.

Bryce, Sampath and Memon [44] discuss the development of a single model and test prioritization strategies for dealing with event-driven software. Since the software changes state based on incoming events, it can be difficult to create efficient test suites that maximize the coverage of all the important test cases. An empirical study showed that two event-driven software examples, GUI and web-based applications, show similar behavior when using the newly developed single model. The idea is that similar subclasses of software types can be cast into a single model and studied together to aid in creating efficient test suites. Various criteria, specific to the event-driven software subclass are run on the parameters of the applications to measure their response. Of particular interest is the evaluation method for the prioritization criteria.

In another paper about improving test case prioritization, Carlson, Do, and Denton [45] discuss using a clustering approach in an industrial case study. The study involves reducing the total cost of regression testing by exploring the underlying relationships in the data, such as data correlations and patterns, to obtain information that would augment regression testing techniques. New prioritization techniques are implemented that incorporate clustering, code coverage, code complexity, and historical data on real faults. The initial results show that the new prioritization techniques can improve the effectiveness of regression testing. Methods for measuring code complexity and prioritizing test cases using code coverage are relevant. The complexity is found using a combination of lines of code and dependency relationships.

Czerwonka, Das, Nagappan, Tarvo and Teterev [46] discuss a Microsoft system called CRANE. The research was inspired by the lack of data on the use of techniques for failure prediction, change risk analysis and test prioritization being used in real-world systems. CRANE is a system used at Microsoft that leverages existing research for developing and maintaining the Windows Vista operating system. The paper describes the design of CRANE and validates its usefulness and effectiveness in practice. This paper is a good resource for learning about how such a system could be developed and implemented for use with a real-world application. The lessons learned from the case study include: data should be simple to understand, empirical and insightful, data needs to be project and context specific, metrics should be non-redundant, information should be actionable, and to have at least one statistical expert in the team. Some areas where CRANE could be improved are the performance of metric collection and the efficiency and effectiveness of the recommendations.

Jacob and Ravi [47] discuss their project for detecting code defects using test case prioritization rules. The project involves data mining techniques to locate two major types of defects in source code in a single process. The two defects are rule-violating defects, like a missing brace, and copy-paste defects, such as extra space after a paste. The proposed detection method for locating defects extracts implicit source code requirements after analysis and uses them to search for defects. The detection method first uses frequent pattern matching to detect rule-violation defects and then creates data-sets of the functions in the source code. These data-sets are run against the defect location methods and the defect report is generated. The results showed that a large number of implicit and undocumented programming rules could be extracted from the source code. Of importance to this research was the notice that between 88.3 and 96.7 percent of programming rules involve variables. This reinforces the choice to make variables the key component of the prioritization process.

In an effort to explore the creation of adequate test suites for integrated test case prioritization for fault localization, Jiang, Chang, and Tse [48] report and empirical study. The study investigates the integration of test case prioritization and fault localization with a postmortem analysis approach. It was found that test adequacy criteria is still lacking in the current state of the research. Program debugging often uses only partial test results. This means that there needs to be some system for selecting and executing certain test cases before others, and determining whether the executed test cases are actually useful to the debugger. The test adequacy criteria is important because they define how much testing is needed before the system is thoroughly tested. It is possible to continue testing for a seemingly infinite amount of time and though random testing is useful for stress testing, it is not useful in all cases.

Following up on [48] Jiang, Zhang, Chan, Tse, and Chen explore the relationship between test case prioritization and statistical fault localization and how well they integrate. In [49] the previous methods are further tested to see whether the effectiveness factors of test case prioritization, such as strategy, coverage granularity, and time cost, have consequences on the effectiveness of fault localization. An experiment was run that included sixteen test case prioritization techniques and four statistical fault localization techniques. The Siemens suite of programs was tested, in addition to grep, gzip, zed, and flex. With the use of test case prioritization up to forty percent of test case executions could be saved, significantly affecting the fault localization. By decreasing the test suite in size any further lead to a serious drop in effectiveness. It was determined that of the factors discussed previously, coverage granularity was the only factor that was not a significant factor.

Wei, Rashid, Pattabiraman and Gopalakrishnan [50] compare the effects of intermittent and transient hardware faults on programs. Of importance in this research is the fault model that is used to represent intermittent and transient faults. We utilize transient fault injection simulation to measure the failure rate of variables under certain conditions. The differences between intermittent and transient fault injection is dependent on the length of the intermittent fault, the fault type, and the origin of the fault in the hardware.

2.3 Software Reliability

In this section we will look at the state of the research in modeling and predicting software reliability and assessing risk in software systems. A review will focus on research in machine learning techniques, software reliability models and methods for predicting the reliability of software.

2.3.1 Machine Learning

Over the last two decades there have been a large number of empirical studies to measure the usefulness of software design metrics on predicting the fault-proneness of modules in software systems. The studies, [51] and [20], showed that coupling between object classes and lines of code were the greatest indicators of the fault-proneness of a software module. It is hypothesized that a class with a higher line of code count than others in the same software project will be more prone to suffer from faults, but further testing needs to be done to compare the results across different software languages. The Chidamber and Kemerer metrics are investigated in another study, [52], that utilizes regression and machine learning methods to detect faults specific to classes in a system. These object-oriented metrics are utilized again, [53], to investigate the quality of software by using Receiver Operating Characteristics analysis. It is shown that quality machine learning algorithms, such as the random forest and bagging, are useful for predicting quality, especially in object-oriented languages. The idea of predicting the quality of software is explored, [54], utilizing the NASA Metrics Data Program, to show that the performance of the predictive models varies more as a result of utilizing different metric groups, and not from different algorithms. It is also seen that a combination of code and design metrics is better than just one or the other.

The classifier has a small impact on the performance of the predictive model when dealing with empirical analysis of software metrics, [55], and researchers gathering the metrics are the greatest influence of the prediction rate. Metrics based on the historical data and the reported number of errors are used to train an Artificial Neural Network and Support Vector Machines to determine the fault-proneness of software metrics in one study, [56]. The metrics found to be the most crucial are determined by sensitivity analysis and are then used as the basis of the predictive model. A literature review on software fault prediction, [57], was done which determined that most machine learning done for fault prediction uses method-level metrics. It is suggested that more practical metric data is used for predicting faults and Naive Bayes is identified as a robust algorithm for this problem.

A study on choosing software metrics for detecting defects in a module, [58], used a variety of feature selection and classification techniques to analyze a collection of software metrics. It was found that in this case study, over eighty-five percent of the metrics could be eliminated and the resulting prediction of software quality would either stay the same, or even increase. Another study, [59], also takes into account how many metrics are needed for getting a realistic prediction of the fault-proneness. This is done by utilizing an intelligent feature selection algorithm for locating predictors likely to improve the models performance. An effective predictor can be built with as little as three metrics. A paper on the current state of software metrics research, [60], showed that there is a large basis of research, but that many of the most cited empirical studies are built around either invalid metrics or suffer from a lack of context when evaluated the metrics. A paper that discusses this misuse of metric data sets, [61], further details why it is important to consider the context of the metrics that are being evaluated when attempting to create a predictive model for fault-proneness. The study concluded that researches must consider the context of how the data will be used, the potential for erroneous findings due to repeated data points and incorporating lower level and commonly used metrics to prevent creating repeated data points.

An invited talk by Aditya Nori [62] explores the use of machine learning techniques for evaluating software reliability. The use of support vector machines, SVM, and linear regression are discussed with respect to calculating good invariants that are useful for program verification. The use of SVMs can be used to determine relevant predicates for predicting reliability. By using machine learning techniques it is possible to minimize superficial non-linearities, meaning unforecasted outcomes that are not necessarily due to trends in the predicates taken from the software but occur without explanation. With machine learning, it is possible to identify the trends and relationships between the predicates and the output of the model, the underlying linear structures for prediction can be identified. Cross-validation is evaluated as a method for addressing bias-variance tradeoffs in prediction results. Cross-validation was found to give more effective predictions for locating faults.

2.3.2 Modeling and Formal Methods

Barry Boehm's work with creating a model for estimating cost, effort and schedule for software projects is discussed in [63]. Also known as COCOMO 81, this model was based on a study of projects completed at TRW Aerospace where Boehm served as the Director of Software Research. These projects utilized the waterfall development model and consisted of anywhere between 2,000 and 100,000 lines of code. COCOMO laid the groundwork for a large aspect of software engineering research dealing with analyzing software projects at varying levels of detail. COCOMO works by using a three-tiered method of evaluation consisting of a basic, intermediate and advanced review process. The basic evaluation was used to calculated effort and cost, based on program size. This was useful for quick estimates of software costs, but ignored many other aspects of development that have an effect on the cost of software development. The intermediate evaluation incorporates some of these, introducing complexity, reliability, hardware attributes, personnel information, and project data. Each of the attributes used in the review is rated and used to calculate effort. The advanced evaluation takes the intermediate process and applies it at each step of the software engineering process. Doing this advanced evaluation of effort gives a measure for each phase of the development process: planning and requirements, system design, detailed design, module code and testing, integration and testing.

Imsand, Evans, Dozier and Hamilton present the concept of utilizing genetic algorithms for vulnerability analysis of national missile defense simulation software in [64]. Genetic algorithms are used to complete boundary analysis and parameter optimization on the target software. This analysis was done to illustrate how different parameters in the software affected heavily parametrized missile defense simulation system performance. Relevant results from this work was the discovery that many of the parameters ultimately had no real effect on the outcome of the simulation. It did not matter what values were used for some particular parameters in the simulation, the output was not affected. The goal of this work as to determine which parameters had the greatest impact on the simulation output. By utilizing genetic algorithms, the authors are able to significantly cut down on vulnerability analysis time. By focusing only on the most influential parameters, the overall analysis process becomes more efficient. Similar results were seen through the fault injection testing in the presented work. Despite injecting faults into particular variables, some had no effect on the execution flow of the program and were thus removed from consideration for predicting relative importance.

A set of extended software metrics for measuring software reliability is explored in [65]. Metrics are proposed that are based on Nelson's software reliability model. These metrics are focused on measuring the software reliability through the view point of the user and focus on moving away from a binary measure of either success or failure with respect to software reliability. The reliability of the software is estimated using an extended adaptive testing strategy. The testing utilizes historical information to limit the number of tests according to a testing budget. The data from the study indicate that testing done in this way were closer to the actual reliability of software than a random testing method. The overall variance of the results using this testing method was lower than the compared methods, random testing and operational profile-based testing. The study found that the proposed method of utilizing historical information and adaptive testing to guide the selection of test cases led to more accurate and descriptive evaluations of software reliability.

A safety case framework to determine reliability numbers for software systems is proposed in [66]. There is a lack of consistency with quantitative evaluation of software reliability between different countries and formal justifications for high reliability figures. The current state of the art for quantifying software reliability uses conformance to accepted standards and statistical testing. This framework utilizes a combination of analytical and Bayesian approaches to use all available information from validation and verification testing. The focus of the work is on investigating how each feature from a piece of software affects the probability model and how the features relate to each other. The probability model is combined with development models and available fault data for estimating residual faults in the software. The primary metrics used in the framework are the probability that the software has faults, the probability distribution for number of residual faults, the failure rate of the critical digital systems from software faults and the conditional probability of a common cause of failure. Using these metrics, the framework develops the map of evidence, structure of the Bayesian belief network model, quantifies the model and interprets the results. Testing is still being done on the underlying theories and tools that the model is based on.

In order to minimize the number of tests needed to accurate determine the reliability of software [67] uses stratified sampling. As opposed to the traditional method of simple random sampling based on the operational profile, stratified sampling splits the population into subsets and randomly samples each subset separately. This method sees an overall decrease in variance and increase in accuracy. The work discussed introduces the traditional way to determine the number of tests needed to demonstrate discrete software reliability. Then the stratified sampling method is analyzed and sample software is tested to show how the effectiveness compares to the traditional methods. The stratified sampling method attempts to provide some concrete evidence that software reliability demonstration testing can be valid for use in real world situations. The stratified sampling method was found to provide a minimal number of test cases while maintaining the accuracy of more traditional methods.

The relationship between coverage metrics and reliability is explored in [68]. It is assumed that code covered during correct execution of the program will also produce the correct result for future test cases that cover the same code. The future reliability tests are built up over time and the assumption that the code covered by earlier tests functions the same for future tests allows for the growth of the software reliability as a function of coverage to be modeled. The failure rate of the software is estimated based on the test effectiveness and the number of bugs found in the software. The paper provides evidence to support a relationship between coverage and reliability that is best used to quantify the relative performance of different coverage metrics on a software system. Another paper on software growth models [69] looks instead at leveraging the reliability growth model to assess the security of the software. The issue investigated focuses on how security problems are rare when running the code under "normal" operational profiles. In the case where security problems are detected in the software, reliability growth models can be used to assess the security reliability. A Poisson distribution is done using the unique security problems that are reported, verified and fixed for non-kernel components. The bug metrics are used to predict the impact of the security problems on the discovery rate of bugs over time. It is found that the rate of discovering security problems is relatively constant when running the code under normal circumstances and in some cases, it may be possible to predict the number of remaining security problems based on reliability growth models.

2.3.3 Reliability Prediction

A prediction model based on support vector regression is detailed in [70]. The model uses estimation of distribution algorithms to optimize the parameters of the support vector regression model. The proposed model combines an estimation distribution model that maintains the diversity of the population of the data and a hybrid model that uses both support vector regression and estimation of distribution algorithms. This hybrid model is used to predict the software reliability based on data acquired during the life-cycle of the software. A probabilistic model-building generalized genetic algorithm is used for estimation of distribution algorithms to overcome the shortcomings of genetic algorithms. These shortcomings include: poor performance for particular deceptive problems and the difficulty of modeling a large number of variants. The results indicate that keeping a diverse population of training data can improve the performance of the model and that the hybrid model performs better than either of the unimproved original models.

Software reliability models using co-variate information are explored in [71]. Neural network regression is used to estimate the failure rates of software from inter failure times and number of failures. A Bayesian method is used to predict the software reliability based around the available software metrics. The freely available software packages Winbugs and R are used to facilitate the testing of these models. The testing is done on two data sets from software development of a real time command and control system. The metrics are gathered over a period of weeks and consist of failure numbers during seventeen weeks with the system, weekly execution time, failure identification man hours, and failure identification computer time. The unified approach to predicting software reliability was proven to be possible with Bayesian inference for a neural network regression model. The advantage to this work was how simple and fast the procedure for testing the reliability of the system was. Potential problems lied in the functionality of the utilized tools, suggesting that it is possible that some cases require a less generic implementation of the neural network model.

Pati and Shukla proposed a hybrid technique for predicting software reliability in [72]. The existing state of software reliability growth models relies heavily on unrealistic assumptions about software usage and are dependent on the environment where the software is run. The accuracy of these models is often questionable when applied to real case studies, despite promising results from theoretical test cases. Pati and Shukla utilize a time series approach to predicting software reliability. An ensemble method referred to as hybrid ARIMA is utilized to predict software reliability based on real world data about software failures. The prediction performance of ARIMA and hybrid ARIMA models is compared. ARIMA models utilize artificial neural networks in a feed-forward configuration along with a supervised training approach to facilitate the training of the model. The hybrid model works by using the ARIMA model to analyze linear patterns in the model and then a nonlinear model is used to analyze the residual patterns in the data. Testing showed an increase in accuracy in the hybrid model over the traditional ARIMA model.

Kapur and Pham propose two general frameworks for deriving software reliability growth models based on a non-homogeneous Poisson process in [73]. The research is done with respect to the existence of imperfect debugging and error generation. A software reliability growth model looks at measuring the reliability of a software system based on the observation of failures and the fault removal processes. It is possible that a development team may not be able to completely remove a fault when they observe a failure and some aspect of the original fault may remain. This aspect of dealing with residual faults is called imperfect debugging. When the original fault is replaced by another fault when attempting to fix the original fault, it is called error generation. The proposed framework is developed for the case where the observation and removal of failures is a single testing process and the case where they are different processes. The complexity of a fault is found to be higher the greater the delay between identifying the failure and removing the fault that causes the failure.

Reliability analysis of a software system using path-based software reliability prediction for complex component-based software systems is discussed in [74]. With the increasing size of software applications, the traditional methods for evaluation software reliability are insufficient when dealing with inter-component interactions in a modular system. The proposed adaptive framework incorporates path testing into reliability estimation to reduce the number of test cases when evaluation modular software systems. The primary metrics for evaluating the software are sequences, branches, and loop structures. Using these metrics, the path reliabilities are calculated and used to derive the reliability of the system. The results of the framework testing indicate a high correlation between actual software reliability and the simulated path reliability. Two real case studies are reviewed using the adaptive framework to show a promising method for estimating the reliability of a modular software system.

A survey of computational intelligence approaches for predicting the reliability of a software system are reviewed in [75]. Two kinds of investigations are done on computational intelligence. The paper provides a systematic review of software reliability prediction studies which consider a wide array of metrics, methods, and techniques. Some of these include: fuzzy logic, genetic algorithms, and neural networks. The second aspect of the investigation involves prediction and data collection using available tools. Three classifications of software reliability models is given. These models include: software reliability growth models, models in stable reliability and constant failure, input domain based models, early prediction models, architecture based models, and hybrid black and white box models. The Metrics for Object-Oriented Design suite is evaluation for its effectiveness at predicting faults and reliability in software. Overall, it was found that computational intelligence approach are more accurate than traditional statistical techniques for predicting reliability.

Zhao, Gu, and Ma present a novel software assessment approach that utilizes neural networks in a network environment [76]. The Fedora Core Linux operating system is the focus of the case study presented for the assessment. Of interest is the application of a neural network in order to evaluation the effect of each component in the software on the reliability of an entire system. Also, the consideration of the system being under an open source development paradigm. It is important to consider many aspects of the software system being evaluated when predicting the reliability. The metrics of interest are programming path, size of each component, skill of the fault reported and other factors. The assessment approach utilizes non-homogeneous Poisson process models and neural networks to improve conventional methods that use software reliability growth models. This is done by calculating weight parameters for each component in the system and then estimating the reliability for the entire system. In order to achieve these weights, it is necessary to consider the effect of a debugging process for an entire system, in addition to the inter-component metrics. It is proposed that the data presented here is indicative of the neural network based approach is effective for open source software users to assess the reliability of the software by using the data sets from a bug tracking system. An emphasis on controlling the development process in terms of reliability, effort, and version-upgrade time, limits the approach by requiring additional work from the developers and knowledge of the software in the development phase.

Chapter 3

Framework Architecture

3.1 Overview

The goal is to create a framework that completes reliability verification and risk assessment on open-source software as efficiently as possible. Compared to the process from [1] there have been some changes to the overall process for locating high priority locations for placing error handling. The overall process remains the same, but there are additional steps involving the use of machine learning algorithms to predict priority functions and variables at two major steps in the process. In addition to the inclusion of machine learning elements, the static and dynamic analysis elements have been expanded to include a mixture of currently accepted metrics and some metric specific to this dissertation. Lastly, the fault injection framework code was updated and expanded upon, refining the injection method and including a larger number of available injection types. The research goals of this dissertation were to expand the static and dynamic analysis portions of the process to increase the number of metrics that could be used to predict the priority of functions and variables. Machine learning is integrated into the process with the goal of making the process for ultimately identifying key locations for error handling more efficient. The fault injection framework is used to validate the predictions and increase the training data sets for teaching the model. Ideally, the process would not initially utilize fault injection, but would instead have enough data to predict the importance of the variables without needing the failure rate, spatial impact and temporal impact. The overall process is shown in Figure 3.1.

The first step of the research is to identify key pieces of open source software that can be easily obtained, compiled, and tested. At this stage of the research, the process does not consider the functionality of the target software or its general structure. The main priority

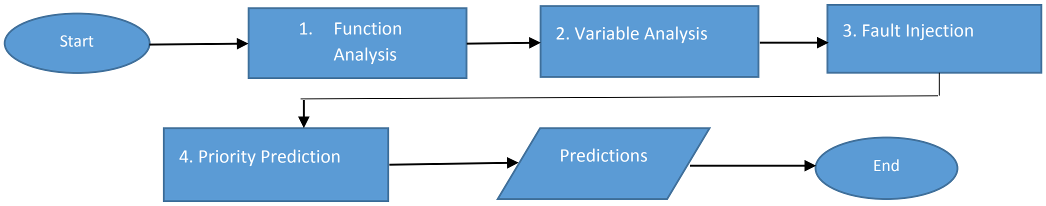

Figure 3.1: Overview of Process

is to validate that it is possible to effectively utilize machine learning algorithms in the context of making reliability verification testing more efficient. Once the target software is obtained, the function metrics are gathered and are sent to the binary classification model which predicts whether the functions in each model are important, i.e. that they are part of the critical path. Once the predictions have been made, the priorities of the functions are calculated and the critical path is determined. The idea behind the critical path is discussed in more detail in [1], but it is essentially the key data flow of information throughout the system across modules and the functions within those modules. This is done by determining the highest priority function and passing that information along to the second stage of the process. This stage will determine the critical path using static analysis to map out the flow of data in the program to locate all the functions that have an effect on the variables in the priority function. Further analysis is done to complete the dataset for variable metrics in this stage. The metrics are sent to the regression model in order to predict the importance of each variable relative to the usage of the target software. Lastly, fault injection is completed on the target software to obtain the relative importance and the results of this testing are used to create a validation dataset. This dataset is tested against the previous predictions in order to validate the effectiveness of the prediction models. The following sections further detail the effort that goes into each stage of the process.

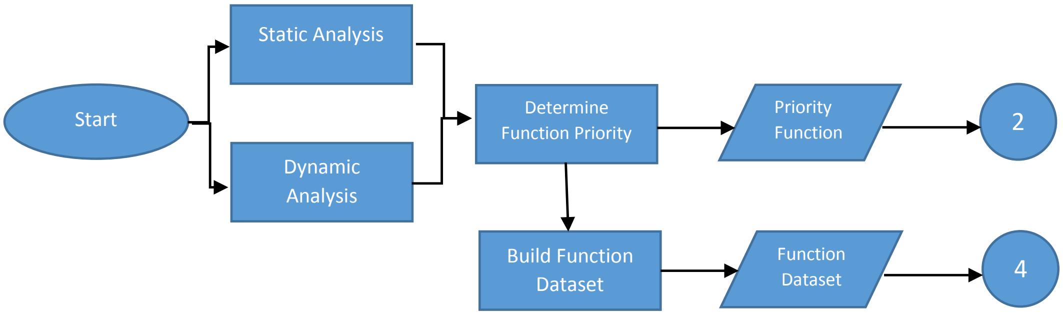

Figure 3.2: Function Analysis

3.2 Function Analysis

The first major step in the dissertation is to evaluate the functions of the target software in order to predict the priority of each function. This is done by a combination of static and dynamic analysis in order to obtain as much key information about the software as possible. First dynamic analysis is done by utilizing code coverage obtained through the same relative use cases that will be used during fault injection. This is done in order to locate where the software spends most of its time, or none of its time, during the intended use cases of the software. This is done by utilizing the gcov tool [77] for software compiled through gcc. The gcov tool is available as part of the GNU compiler collection suite. This is a well documented and testing tool that is effective for testing variants of the C language, but does further limit the current implementation to testing against C source code. The output of the tool is a .gcov file that is a copy of the source code with a number next to each logical line of code that shows how often the line of code was executed since the .gcno file was created at compile time. A few examples of this output are included in Appendix B. The output of the gcov tool is used to obtain the frequency that the functions in the software are executed. This frequency measurement is combined with the logical lines of code metric found in the static analysis to calculate the probability that a randomly generated transient data fault will occur during the execution of a line of code within each function. This is illustrated in equation 3.1, where $ELoC_{\mathrm{f}}$ is effective lines of code. The lines of code for a function, LoC, was determined through code analysis and the function call frequency, F_{cf}, was determined through code coverage.

$$ELoC_f = LoC * F_{cf} \tag{3.1}$$

After calculating the effective lines of code for each potentially high-priority function, the probability that corrupted code is executed in that function at any given point during run-time is found by equation 3.2. $ELoC_{\mathrm{f}}$ is the effective lines of code for the function in question and $TELoc$ is the total effective lines of code across all functions.

$$P_E f = ELoC_f / TELoC \tag{3.2}$$

Instead of assuming an equal chance across all functions and variables for a transient data fault to occur, a probability of corruption is determined in order to relate the data in terms of how likely a randomly placed bit fault is to occur in memory belonging to a particular function or variable. At this stage of the research, the static size of the code artifact is used in order to create this relation. The static size of the functions is determined by the difference between the start of the function object location in memory and the start of the next file object in memory after that. The nm [78] and objdump [79] tools are used in order to obtain the file object information about the size of each of the functions. This size value is used to calculate the probability that a randomly generated transient data fault would occur in a bit located in memory belonging to the code for that function. This probability of corruption for each function, P_{Cf}, is calculated as shown in equation 3.3, where Mem_{f} is the memory usage for that function converted into decimal and $TMem_{\mathrm{f}}$ is the total memory used across all functions in question.

$$P_C f = Mem_f / TMem_f \tag{3.3}$$

The values of the probability of corruption and execution together indicate the probability of a transient data fault occurring in a random location during a random point during code execution and the resulting corrupted data being accessed at some point during the remainder of execution. The two key events: a transient data fault occurring in a bit in memory and that memory being accessed during execution, are independent and so the probability of both occurring and potentially causing the program to enter an undesirable program state is calculated by multiplying the values as seen in equation 3.4.

$$Priority = P_E f * P_C f * p \tag{3.4}$$

This probability value is then used along with the propagation value, p, for each function in order to measure the priority of each of the functions. The propagation value, as described in [1], is determined by utilizing Doxygen to create data-flow diagrams of each function in each module in order to determine the potential for the impact of a data fault to spread to other functions. Doxygen [80] is run against the complete source code of each piece of target software and utilizing the dot library of graphviz [81] in order to create the graphs of the data-flow across the program. The priority metric for each function is calculated as shown in equation 3.5.

The other key factor during this stage is the static analysis that is done to the functions in the target software. In order to measure the capability of currently accepted code metrics to predict the priority of functions and variables in the target software the Halstead [82] and McCabe [83] metrics are used. These metrics are used as the features in the binary classification model in order to predict whether or not each function is important. After evaluating each function for its relative priority within the target software, the highest priority function is passed along to the next stage of the process in order to complete the variable analysis on the target software. It is important to note that at this point, the dataset of function metrics has been sent along to the fourth stage of the process in order to get the predictions. For this dissertation, the process moves along utilizing the techniques discussed in the next

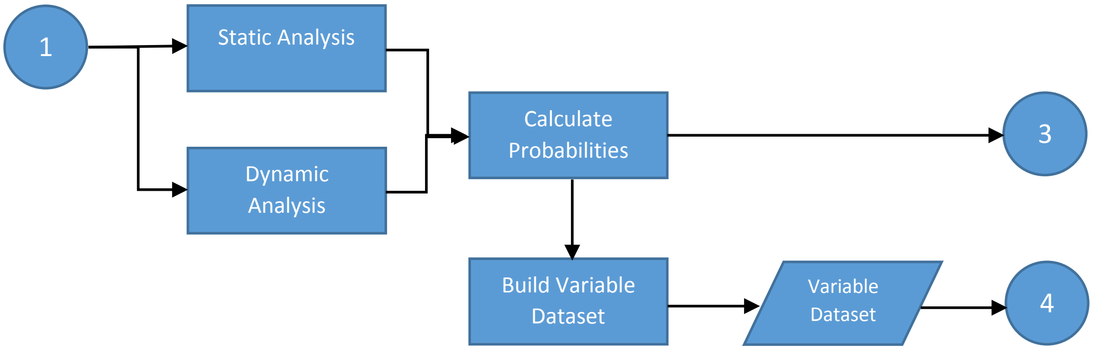

Figure 3.3: Variable Analysis

stage and fault injection in order to obtain the importance of the variables, and the data gathered and sent to the machine learning models is used to validate that the models can be used in order to make the process more efficient. This will be seen in the validation test shown in the next chapter.

3.3 Variable Analysis

Once the highest priority function is determined the analysis of the variables in the target software can begin. The first step in this stage of the process is to utilize dynamic program slicing on each of the variables in the high priority function in order to determine what variables potentially have an effect on the value of the variable in question. This means all variables that are involved in writes to the variable, any variables in branch statements that can alter the value stored in the variable, and any variables that effect variables that effect the variable in question and so on. This process is recursively completed until no more variables are left to add to the dataset. At this point, static analysis is completed on each of the variables in the dataset in order to obtain as much information as possible that can be used when predicting the importance of these variables. These metrics in the frequency that the variable value is accessed during program execution, using the code coverage output from gcov, the number of variables that the variable is affected by and affects, the number of reads and writes for each variable and the number of other functions that the variable connects to in the flow of data. These static metrics are combined with the code metrics for the function that the variable is located in.

Dynamic analysis is done in order to obtain the size of each of the variables. This is important because we are simulating the random occurrence of a bit fault in memory. Knowing how much memory is needed in order to store the value of the variable tells us how likely a randomly generated fault is to occur in memory belonging to a particular variable. There are some established variable sizes for each programming language and operating system, so it is possible to make an educated guess as to the size of the variables in the target software, but it is more accurate to directly instrument the code to log the sizes of the variables within the code. This is necessary because the sizes of variables is dependent on the implementation of the system that the system is compiled and run on. Also, it is necessary to consider variable types that are nonstandard and enumerated types that will be specific to the software being tested. The framework utilizes probes and a variable log in order to track the value of variables. In the case of this dissertation, all pointer sizes and values are measured based on what is stored in the located where the pointer is indicating, and not the pointer variable itself. This means that for a float pointer, the size is measured as the size of a standard thirty-two bit floating point variable.

After collecting all of the desired metrics through static and dynamic analysis, the probability of execution and corruption for the specific variables is calculated. In reference to the variables, the probability of execution, $\mathrm{P_{Ev}}$ is calculated by taking the frequency the variable is referenced in the code coverage, $\mathrm{F_v}$, and dividing it by the total frequency of all the variables in the code coverage, $\mathrm{TF_v}$, as indicated by equation 3.5.

$$P_E v = F_v / TF_v \tag{3.5}$$

With the probability that a variable containing a data fault will be executed relative to the other variable in the system is known. In order to determine the probability that a data fault will occur in the memory space for a variable at a time when the variable is being accessed, the probability of corruption must be determined. These variables are similar to those found for each function, instead using frequency and size values relative to the variable. The probability of corruption for a variable, $\mathrm{P_{Cv}}$, is determined by taking the size of the variable, $\mathrm{Mem_v}$, and dividing by the total size of all variables in the system, $\mathrm{TMem_v}$. This is shown in equation 3.6

$$P_Cv = Mem_v/TMem_v \tag{3.6}$$

$$P_Fv = P_Ev * P_Cv \tag{3.7}$$

As with the probability of the fault occurring for the function in the previous section, the $\mathrm{P_{Fv}}$ is determined by taking the probability of two independent events, $\mathrm{P_{Ev}}$ and $\mathrm{P_{Cv}}$ and multiplying them together as in equation 3.7. This value becomes important for calculating the relative importance of the variables in the system as it provides context to the importance metric based on the probabilities that a data fault would occur at a time where the artifact is being accessed. The process for evaluating the relative importance of the variables has been adjusted from [1] to bring the calculated more in line with actual results found through fault injection testing in the next section. At this point in the evaluation process a dataset of the variable metrics is constructed and sent to the regression model in order to predict that importance values of each of the variables in the critical path. These predictions are then validated by completed fault injection testing on each variable to measure the failure rate, spatial impact and temporal impact. When applied to a real-world open source project that is going through reliability verification, it would not be necessary to complete the fault injection as the goal is to obtain the relative importance of variables to determine where to place error handling. The output of the regression model would provide those values and the user can determine how many to wrap, as discussed in [1].

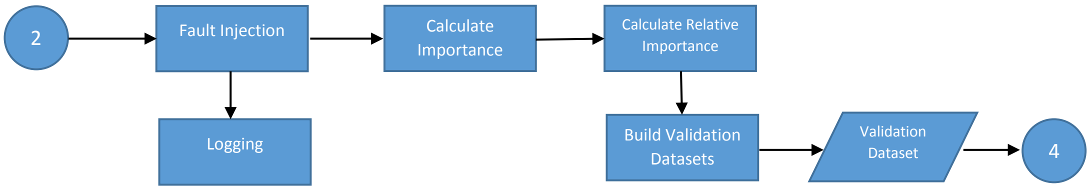

Figure 3.4: Fault Injection

3.4 Fault Injection

The first thing to consider when doing fault injection testing is the fault model that will be used. Depending on the type of fault that is going to be considered, the type of injection will change. The type of fault model used has two major parts: a local model and a global model. The local model states the types of faults that could occur in the system and the global model constrains the occurrence of the faults from the local model, so that reliability can be measured. The faults considered were those that could occur at any time or at any place during the execution of the system. The fault injection framework simulates the occurrence of transient data faults. An example of one such fault would be a UAV exposed to extreme temperature changes that cause the data stored in some part of memory to become corrupted. This does not include permanent hardware faults, like the memory suffering physical damage and the data being lost altogether. The local model is a transient data value failure, which means that any variable can be corrupted at any time and could be corrupted more than once. The global model is the assumption that any variable in the system could be affected by this data failure. [1]

Some changes have been made to the implementation of the framework in order to reflect some fundamental changes to the fault model. In the previous work the intended usage of the framework was to cause system failures and log how many failures occurred for each variable. This way of testing was done in order to determine which variables had the greatest likelihood of causing system failure and how big of an impact they had on the system. This is an important measurement, but for the purposes of this dissertation it was a limiting way of conducting testing. The goal of the research is to allow for the efficient testing of open source software for reliability verification. With the focus being on creating relative use cases and a stub program to utilize the target software in as accurate a way as possible. It is necessary to consider the case where the open source code will not be integrated into the new system without undergoing some changes. With this in mind, it becomes necessary to measure the effect the variables have on the flow of the program given these relative use cases. For this dissertation the focus is not on full system failure, but instead on situations where the target system enters an invalid program state.

In order to test for this, a golden log file is created that contains the list of steps through the program during execution that map out the intended flow of execution through the program. When the target software is run using the fault injection, the execution trace is logged and compared to the golden run of the target software. Any time the injected fault cause the program to go off of the expected program flow, that is determined to be a failure. Fault injection testing done in this way allows the framework to locate faults that cause a system failure as before, causing the program to crash and creating an invalid output due to the injection causing errors in the program will still be caught this way. But, now the failure rate will be indicative of a variable to cause the target software to enter an invalid program state that may not necessarily cause a system failure in the form of a crash, but creates a situation where there is a high risk. Testing in this way allows the functions in the target software to be considered relative to the other functions in the system, but also to get information about the effect this code would have if taken out of context and put inside of another module. In addition to allowing a more complete view on the state of the system during fault injection, this can further highlight the areas that need further error handling by showing exactly where in the program a variable with a high importance was going off of the expected execution trace.

3.4.1 Framework

Injections

The injection functionality is incorporated into the system first by adding the injectors.h header file to all file to be tested and by placing the injectors.o object file in the compile path for the system. Once this is done, it is possible to inject errors directly into variables at any location in the code. There were two different types of injection functions written for the framework. The first takes a specific value and inserts it into the variable in order to test boundary conditions or specific faults, such as NULL or zero. The manual injection function takes in four inputs. The *iid* variable is the string that identifies the injection point. This value is compared to a file containing all the probes and injections that will be used in the current experiment. If the string is located in the file, then the injection is completed and the specified injection value, *inject_val*, is returned. If not, then the original integer value, int *var*, will be returned. The variable *p* specifies the probability that the injection will occur.

The second function, which is the primary function used for this dissertation, injects a random error in the variable. This is done by flipping a random bit in the variable and then returning variable with the flipped bit. The prototype and way that the function is called is very similar to the manual injection function, but instead the injection value is randomly defined from within the function. The handling of flipping the bits in a single variable is dependent on the type of variable that is being considered for injection. The general process is done by casting the incoming variables address to an unsigned long* and dereferencing it. The bitwise operation is completed on the unsigned long, which should be the longest variable type available on the system so that it can be used to represent any other variable including a double. When the single random bit-flip is complete, the variable is then cast back as the original variable pointer and then dereferenced.

There are exceptions to this case, such as when the variable to receive the fault injection is greater than 32 bits in size or when dealing with pointers. Appendix A lists the source code for the framework and it can be seen that in order to handle longer variable types it is possible to keep the general strategy for injections, but utilize unsigned long long variable types. Also, when dealing with pointers it is necessary to save the value of the located pointed to by the pointer in an appropriate variable type, complete the random bit flip, and then reassign the pointer to the new variable. This essentially changes the value of where the pointer is looking, so it becomes necessary to save the original value off in the case of an injection, to maintain accurate logs of the variable state.

Instrumentation of a system involves a single call to an initialization function for the injectors which includes a seed function call for rand and then the injector functions themselves wherever the user wishes to place a fault in the system. [1] The code coverage output for the target system is used in order to locate the ideal location for the injection to be placed. An areas in the desired function that is accessed during regular execution of the code is targeted for injection and the result is monitored by the framework logs.

Probes

The probes currently serve the purpose of allowing the tester to see if the variable injection was completed and what the value of a particular variable is at a given location in the system. There is only one set of probe functions, where each function handles a different type of variable probe. Each time a new type of injector is added a matching probe function is added in order to log the changes in the value of all the possible variables. The probe functionality also has to check the experiment list to see if that probe is to be used during the current experiment. This allows all of the injections and probes to be placed into the instrumented system at one time and then by modifying the experiment list document, the framework knows which of the injection and probe sites to use for that particular experiment. The probes also play an important role in determining the spatial and temporal impact of the faults within the system. By ”turning on” all of the probes in the functions when completing an injection test against a variable, it is possible to monitor what variables have a value different than the expected value from the golden run and how long that value in that variable stays incorrect.

Environment Simulator

The environment simulator is the program that is actually run when testing the instrumented system. The program takes in the experiment number as a command line input and then runs the intended use cases of the function a set number of times and collects the results of these experiments in a file in a results directory with experiment number labels for each file. This is done in order to streamline the process of testing the instrumented system, but also to simulate the intended use of the system. Any inputs the instrumented code may need are controlled by the environment simulator and any outputs are dealt with accordingly. This way, the testing of the instrumented component can be done in a way that closely represents how it will fit into the entire system. [1]

The environment simulator is written to replicate the intended usage of the open source component; consequently, it will differ based on each system. The way that the environment simulator is written to test one open source component is not necessarily the same way the simulator will be written when testing a component for another system. The general method is the same, but the specifics of implementation will be different. It is important to note that this dissertation attempts to accomplish a proof of concept of a task utilizing C code and not a means for tested all types of code.

The testing process was driven using bash scripts as opposed to creating a C stub just for testing the target software. Eventually this functionality will be changed to be primarily script driven to minimize the effort needed to simulate the environment where the target software will be executed. It is much more efficient to utilize scripts than to write a stub function in C for each piece of software. To create a more optimized and automated framework, the simulation of the target software could be driven by a more flexible scripting language, such as python.

3.4.2 Logging

The framework utilizes logging at various stages of execution in order to more efficiently conduct the injections and verify the results. The first part of the logging functionality is utilized through the injector and probe sections in order to monitor the value of particular variables throughout execution of the code. This is done primarily for testing for the spatial and temporal impact of the variables given an injected fault. The environment simulator logs the output of the target software for each iteration of the testing phase for troubleshooting potential problems in the proper execution of the framework or target software. The last element of the logging functionality is to track the execution trace of each iteration of the target software. In order to track the number of failures in the system, each execution of the target software has the list of steps logged. These logs are diff'd against the golden execution traces in order to look for executions where the expected flow of execution in the program was altered. This means of logging the execution steps throughout the code is done by utilizing bash scripts to automate the gdb [84] debugger for the gcc compiler. A script is written to add functionality to the gdb debugger in order to automatically step through the execution of the software to create the expected flow for each potential use case.

3.4.3 Importance

Given a software system with functionality distributed logically over a set of distinct components, the spatial impact of a variable v, of component C, in a run r, is denoted as in equation 3.8. The spatial impact is then defined as the number of components that are corrupted in r when the variable v is corrupted.

$$\sigma_{v,C} = max\{\sigma^{r}_{v,C}\}, \forall r \tag{3.8}$$

Spatial impact finds the number of modules affected when a variable v in component C is corrupted. The higher the value, the harder it will be for a system to recover from the corruption. Given the same software system, temporal impact as defined by [85] is the impact of a variable v of component C in a run r, denoted as in equation 3.9. This value indicates that number of time units where at least one component of the software system remains corrupted in r. [1]

$$\tau_{v,C} = max\{\tau^r_{v,C}\}, \forall r \tag{3.9}$$

After determining the spatial and temporal impact for each relevant variable in the critical path, it is important to determine the failure rate of each of the variables. The failure rate is the failure rate determined through fault injection when the faults are injected into each variable given a specific set of use cases for the target software. The previous work did not use just the local failure rate because the importance calculated with just the failure rate of a variable and the local spatial and temporal impacts are limited to just the modules where they are calculated. Local failure rate is used because the method for evaluating failures in target software has changed. This dissertation modifies the previous relative importance metric by adjusting the calculation to put a distinct change between local and relative importance of variables and not losing the balance between the failure rate and the impact of the failures.

The general-form equation for calculating the importance of a variable is defined in equation 3.10, where G, K and L are arbitrary functions that determine the importance, spatial impact, and temporal impact respectively. The value incorporates the failure rate and the spatial and temporal impact of a variable. This is done because a variable v in a component C with a high spatial impact is likely to pass the corruption on to other variables and a high temporal impact indicates that recovery efforts will be less effective. [1]

$$I_{v,C} = G[K(\sigma_{v,C}), L(\tau_{v,C})] \tag{3.10}$$

The final equation found for the importance of a variable is as defined in equation 3.11, where f indicates the failure rate and n and m indicate the focus of the testing. To be more specific, the greater n is the more the metric focuses on the failure rate of the variables and the greater m is the metric focuses on the spread and time of existence of corruption. An m of zero and an n greater than zero would measure solely for failure rate of the variable and conversely, an n of zero and an m greater than zero would measure only the impact of the failure. The importance metric is altered to be a combination of the local importance and relative importance from [1]. This is done because of data gathered during this dissertation that brought up a problem with the current implementation of importance. The variable importance found using the failure rate of the variable in that function and the spatial and temporal impact as seen in equation 3.11.

$$I_{v,C} = [1/(1-f)^n][(\sigma_{v,C}/\sigma_{max} + \tau_{v,C}/\tau_{max})]^m \tag{3.11}$$

A fault that is introduced into a system, but is never executed has no effect on the outcome of the current run of the program. This means that if the fault is injected randomly at any point during the execution of the code, the actual likelihood of the fault being effective for a particular variable is assumed to be P_{Ev}. The likelihood of the transient data fault occurring in the memory relative to a particular variable is assumed to be P_{Cv} These values can be used to define the relative importance as defined in equation 3.12, where σ_{max_r} is the max spatial impact on the critical path, τ_{max_r} is the maximum temporal impact on the critical path, f is the failure rate and P_{Fv} is the probability that a transient data fault will occur in the memory location for the variable, v, at a point in the code where it will be accessed. Relative importance is calculated in this way because it maintains the original aspect of the importance that is based around how the function holds up against other variables in the same function, but is made relative to the rest of the variables on the critical path based around the probability that the random transient data fault will occur at a time and place that effects the variable in question compared to the probability that the same occurs for any other variable.

$$RI_{v,C} = P_F v * [1/(1-f)^n][(\sigma_{v,C}/\sigma_{max_r} + \tau_{v,C}/\tau_{max_r})]^m \tag{3.12}$$

The reason the relative importance is calculated in this way is that the previous method of using the P_{Fc} to only inject given a certain probability drastically reduces the failure rate of each of the variables. Doing the testing in this way requires a greater number of test cases to get trusted results and causes problems when utilizing the importance equation. Since each variable will likely have a very small failure rate, the accuracy of the equation begins to drop and less emphasis is naturally placed on failure rate when the first half of the equation comes out to be extremely close to 1 for each variable. Once the relative importance, $RI_{v,C}$ has been calculated for all variables in a critical path, they are ranked according this value. This process can be repeated for multiple critical paths, where the relative importance is calculated across all paths. With the relative importance calculated, the validation dataset for the target software is completed and can be sent to the priority prediction models to validate the previous predictions. The validation dataset is the same as the variable dataset from the previous step, but with the fault injection results added. To validate the predictions the relative importance calculated using the fault injection framework can be compared to those made by the prediction model.

3.5 Priority Prediction

3.5.1 Function Priority

In order to predict the priority of the functions in the target software system, the features of the software that are collected in the first stage of the framework are given to a binary classification model which determines which of the data points are of high priority. In this case, the features that are extracted from the software being tested are the static

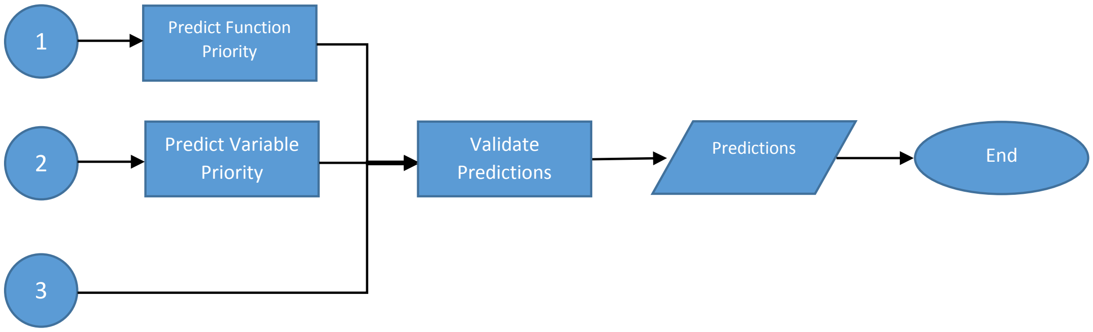

Figure 3.5: Priority Prediction

and dynamic metrics discussed in the function analysis section. These features serve as the attributes that the machine learning model will use to create a predictive model that uses those attributes as input and predicts a value of 0 or 1, where 0 means the function is not important and the variables within will not be considered for analysis and a 1 means that the function is important and lies on the critical data flow path through the system. At this stage of the research, the primary concern is to see how the currently used features could be used to identify trends between the priority of functions for risk analysis.

For the binary classification WEKA [86] [87], a collection of machine learning algorithms for data mining, it utilized to obtain the prediction data. The software was developed at the University of Waikato in New Zealand. The datasets created through the static and dynamic analysis are converted from an excel CSV file to the ARFF file format utilized by the WEKA program for analysis. Utilizing WEKA the datasets are tested using a series of machine learning algorithms and pre-processing techniques in order to obtain the highest possible predictions without overfitting the data. The GUI front end of WEKA was utilized, but in future work it would be possible to use the java and python extensions to allow for the process of collecting the data and making the classification predictions automated. A series of different classification algorithms are investigated in the experiments in the following chapter.

3.5.2 Variable Priority

The variable priorities are predicted by utilizing machine learning algorithms and the metrics gathered through static and dynamic analysis. In order to create the dataset need to determine the predictions of the variable priorities, the static and dynamic analysis data from the second stage of the research are collected and given to the regression algorithms utilized in WEKA to generate numeric predictions for the relative importance of each of the variables. The static and dynamic analysis metrics are used as the features for the regression models and the relative importance serves as the class that the algorithms are attempting to predict the values for. In the case of the regression modeling, the output of the metrics through WEKA are not utilized, but only the predictions are considered. Further analysis must be done beyond the data provided by WEKA in order to fully utilize the predictions made for each of the variables.

Once the predictions have been made they must be ranked from highest to lowest and then compared to the actual rankings of each of the values taken from the fault injection component output. The data is placed into an excel spreadsheet where the actual relative importance values and the predicted relative importance are ranked. With the resulting rankings, the percent error in each ranking is determined and the mean error across all rankings is determined. In addition to these overall prediction metrics, the individual data points are broken down into ten percent increments to identify the highest priority variables according to the ranges that the user is interested in. This allows for some flexibility based on what how much error handling code the user wants to add to the software in order to increase the reliability of the code. This will be dependent on the type of system the target software will be integrated into. For example, a long running piece of software that monitors risks over a period of time may be willing to have some slow down by adding error handling to the top twenty or thirty percent of relative variables, where are program like a flight control system for a UAV may be limited to only the top fifteen percent of variables to prevent too much slowdown in the system execution. The regression algorithms utilized and the process for collecting the ranks is discussed in more detail in the experiment chapter.

Chapter 4

Experimental Design and Results

This chapter looks at three key experiments completed utilizing the dissertation research in order to establish the baseline operation for the designed framework and then to validate the use of the framework on new piece of software. The first experiment looks at utilizing the function metric data with binary classification algorithms to locate any trends in the metrics with determining the high priority functions through finding the critical data flow path through the software. The baseline model for predicting the high priority functions is created and tested using a technique called ten fold cross-validation. The second experiment repeats this process of looking for trends between the static and dynamic metric data and the calculation of the relative importance of each variable in the critical path. A baseline model utilizing the data is created and tested again using ten fold cross-validation. These two baseline models serve as the starting point for the third experiment which is the validation testing for the dissertation research. The two trained models are used on the datasets collected on software that has not been seen before by the framework and predicts the highest priority functions and variables in the system. The fault injection data collected by the framework is utilized in order to validate the results of these experiments. Lastly, the test data is added to the training data from experiments one and two to create one large dataset, and ten fold cross-validation is done again to observe the effects of increasing the size of the training data on the prediction rate.

4.1 Priority Function Prediction

4.1.1 Case Study

The case study described in this section is based on an open-source software project called Mp3gain. It is software that allows the user to perform numerous alterations of .mp3 sound files, such as normalizing gain and adjusting other audio settings. The data was gathered from a series of tests meant to obtain metric data from realistic use cases of the software. The data was gathered under the assumption that another software system was going to integrate parts of the functionality of Mp3gain and needed to pass reliability verification testing on the code before integrating it into the system. The data is context-sensitive to the testing methods described in previous work, [1]. The goal of this study was to complete some exploratory research and to obtain some initial results as to the effectiveness of utilizing machine learning techniques to make the process for identifying which methods and functions in the code have the highest likelihood of containing variables with a high failure rate when introduced to faults in the data.

Of particular interest was identifying trends that would indicate important functions in open-source software and post-release software in general. The hypothesis is that when approaching open-source software in order to use it in a pre-existing system, the trends in the metrics will be different than code that has been put through rigorous design constraints from the beginning of development. This will be evident in the results in the form of outlying data points an order of magnitude higher than the average and for relatively erratic design metric results from function to function. These effects are likely do to the open style of development that generally accompanies an open-source project. The classifications for each of the instances was determined by completing fault-injection testing given the set of intended use-cases and calculating the average failure rate across variables in a function and identifying levels of importance for each function from these values.

4.1.2 Test Setup

A series of tests were completed in the explorer to get initial reactions from a variety of classification algorithms and pre-processing algorithms. Then, the WEKA experimenter was used in order to complete more rigorous tests of the metric data sets once the classification and pre-processing algorithms were chosen. In all test cases, ten-fold cross validation was used in order to get more accurate data from the results due to the size of the sample set, taken from a single project. For the experimenter data, ten iterations of each ten-fold cross validation test was completed. The following sections will discuss the different metrics collected from the software, pre-processing algorithms used, classification models, and cost-sensitive analysis.

4.1.3 Attribute Selection

In order to identify which attributes have the greatest impact on the prediction model, a selection algorithm is used to rank the attributes according to the amount of information obtained from having each metric in the data set. In this case, attribute selection is measured by merit and then ranked. Across a set of cross validation tests, the data set is analyzed to see which of the metrics provides the greatest amount of information, meaning the result of the evaluation criteria across all of the folds of the cross validation. The merit for this nominal classification, important or not, is representative of the error rate contributed by the inclusion of the attribute.

Metrics Used

n1,n2,N1,N2 These metrics form the basis of the Halstead metrics, [82], and represent measures of the operators and operands in the software. The number of unique operators is n1 and the total number of operands is N1. Likewise, the number of unique operands is n1 and the total number of operands is N2. These values are useful for calculating other metrics and for providing raw data about the size of the code, beyond just lines of code.

Derived Halstead These metrics are calculated from the previously determined metrics. The program Vocabulary is measured as the number of unique operators plus the number of unique operands. Program Length is measured as the total number of operators plus the total number of operands. Program Volume is the amount of information contained in the program measured in bits, measured as the program Length multiplied by the 2-base logarithm of the Vocabulary. The program Difficulty represents the likelihood of errors occurring in the software, determined by the unique operators divided by two and then multiplied by the total number of operands divided by the unique operands. The Effort is calculated by multiplying the Volume by the Difficulty and represents how hard it is to understand a program. Lastly, the Intelligent Content of the program is calculated by multiplying the program Volume by the Difficulty and is a measure of the complexity of the code.

Code Line Metrics The modified lines of code contains the number of lines in the software no matter what they contain and the actual logical lines of code in the software is measured as a separate attribute which disregards white space and comments. The preliminary lines of code includes these values and is removed in pre-processing for zero value added. The percentage of code that are comments and branches are also included in the data set. The percent comments was included as a measure of the developers effort to keep maintainable code, but the percent branches in the code is a measurement of the complexity of the code. A high number of branches can quickly lead to a high complexity and code depth.

Complexity and Nesting Depth The complexity of each function is calculated using McCabe's cyclomatic complexity, [83]. It is influenced by the number of if statements, loops, switch cases and returns in the software. The cyclomatic complexity is measured by evaluating the control flow of the program using a series of nodes and directed edges to connect the nodes. The nodes represent indivisible groups of commands in a program. This is done in order to measure the number of potential paths through the code in order to determine how difficult it will be to fully test and maintain the code. The nesting depth is a measure of how many of these change of flow points are located in the software. For example, a for loop would increase the nesting depth by one for its duration. If there is an if statement within that for loop, the nesting depth goes up by one again for its duration, and so on. The average nesting depth is calculated for each function to determine how complex the control flow is in the program in a line by line fashion.

Relative Use-Case Metrics The metrics discussed in this section come from the testing methods use in the previous work, [1]. The Frequency metric measures the number of times that a function was called during the chosen use-cases for each of the testing suites. The propagation measures the call depth of the function in relation to the other functions in the program, not unlike program coupling. The effective lines of code, ELoC, is calculated by multiplying the frequency that a function gets called by the lines of code contained in the function. This metric represents how many lines of code total a function represents during execution of the program. This is important when determining the next metric, probability of execution. The probability of execution, P_{Ef}, is the measure of how likely it is that a line of code that contains a variable with a data fault will be executed, measured by dividing the effective lines of code for a function by the sum of the effective lines of code across each function. The reasoning behind this is that even if a data fault occurs somewhere in memory that effects a line of codes, if the locations in memory belonging to that line of code and variables within are not accessed after the data fault occurs, then the fault is unlikely to become a full-blown failure.

Merit and Rank of Attributes

This section will provide the details of the attribute selection process and will discuss the implications of the results. Table 4.1 shows the average merit and average rank of key metrics. Each of the values is augmented by its standard deviation to provide some context for the results across all folds of the cross validation. This method is useful for identifying which attributes are actually useful in predicting the importance of a function and which attributes do not contribute meaningful data. Any of the metrics that fell below the defined threshold are omitted from the table. At this stage it becomes clear that many of the metrics are redundant or not immediately useful for our intended testing methods and can likely be eliminated. The table indicates that the relative metrics for evaluating functions for importance have the greatest information gain, along with a couple Halstead metrics. It is likely that the percent comments metric was evaluated highly by coincidence based on the erratic nature of the developers tendency to leave comments on different sections of code, which is not uncommon in open-source developed by an individual or small group.

The information gain attribute evaluation evaluates the worth of an attribute by measuring the information gain with respect to the class using the formula that follows, where H is the information entropy. The information entropy is the concept of how much information about the class we obtain by observing the data in the attribute. Each event, or data point, is analyzed and the amount of information located in the attribute is evaluated and given a value for merit. This algorithm does not take into account the interaction of attributes. The ranker algorithm is used in conjunction with these algorithms in order to place the attributes in order according to their evaluated values for entropy, merit, etc.

$$InfoGain(Class, Attribute) = H(Class) - H(Class|Attribute) \tag{4.1}$$

Table 4.1: Attribute Information Gain Evaluation

Average Merit	Average Rank	Metric
0.316 +- 0.048	1 +- 0	PoE
0.316 +- 0.048	2 +- 0	ELoC
0.208 +- 0.106	5.6 +- 5.22	Frequency
0.065 +- 0.1	6.5 +- 5.22	Difficulty
0.078 +- 0.096	11.3 +- 5.57	Propagation
0.053 +- 0.081	14.4 +- 6.53	Percent Comments
0.021 +- 0.063	20.4 +- 4.8	n1

4.1.4 Discretization

Discretization is used on the data set in order to bring the selected metrics into more manageable ranges. There are a number of outlying data points that have certain attribute values that are much higher or lower than the average that skew the data. These outliers cannot just be removed because they are important data points in illustrating the unreliable nature of the development style common in open-source software. The discretization method utilized in WEKA, [88] [89], defines a threshold for the attributes in the data set and essentially places all of the instances of the attributes onto one side or the other. This is helpful in identifying key turning points in the data that shows a general trend in certain attributes that are common among important functions. For example, a complexity of fifteen may be a threshold where the majority of points below fifteen are predicted as not important and the majority of points above fifteen are predicted as important. A supervised discretization method is used when training the model to adjust the numeric ranges of the selected attributes to define clear points in the data that influence the prediction. The classifier is taken into consideration when determining the ranges to be used in supervised discretion, which creates a binary distribution of instances around a single threshold.

This discretization is done for each fold of the cross validation, instead of doing preprocessing on the entire data set. In order to prevent potential over-fitting of the data and getting inaccurate results, each iteration of the cross validation folds is discretized as a stand alone process. This was done using the built-in Filtered Classifier in WEKA which applies a given filter to the data before running each fold through the classification algorithm that has been chosen. This method provided more realistic results than when pre-processing the entire data set since the data set was fifty test cases, it was important to take full advantage of the cross validation process to achieve an accurate result. The discretization pre-processing takes much of the decision making away from the classification algorithm and puts it into the ranges that are determined for each attribute. If the same subset of discretized attributes are used for each classification, they are all likely to build around the same thresholds for the attribute with the highest merit. For example, the J48 decision tree classification method will have a single node that splits around just one attribute, in this case probability of execution.

4.1.5 Classification

A wide variety of classification algorithms are used in the initial testing process, and as indicated by other research papers discussed above, the metrics used and the pre-processing done on the data set had a much greater impact on the outcome of the prediction process than the chosen classification algorithm. The results of each of the classification algorithms, without cost-sensitive analysis is shown in Table 4.2. The more important aspect of Table 2 is that the only thing that changes is the mean absolute error, MAE, and root mean squared error, RMSE, and the predictions. The overall confusion matrix remains the same, because the supervised discretization adjusts the ranges on only the attributes known to be important and leaves the other attributes as a single range. This reinforces the point made earlier about the classification algorithm itself being less important than the actual metrics used and the pre-processing done on the data-set. It is also possible that a larger data set or a different data set may have different outcomes, but for initial exploratory data, the implications are of great interest. Table 3 repeats the same set of tests, but instead leaves out the pre-processing, giving a wider range of results.

Table 4.2 and 4.3 display the resulting evaluation analytics for the data set when run through the given classification algorithms, where MP is the Multilayer Perception, NB is Naive Bayes, and IBk is the K-Nearest Neighbor algorithm. The kappa statistic is a measure of the agreement between the two classifications categories. Mean absolute error is used to measure how close the predicted values are to the eventual outcomes, calculated by taking the average of the absolute errors. The root mean squared error is a measure of the root of the averages of the squares of the errors in the classifier, meaning the root of the difference between the predicted class and the actual class. The true positive rate, TP, are the instances

Table 4.2: Classification Comparison With Discretization

Evaluation	J48	MP	NB	IBk
Percent Correct	75	75	75	75
Percent Incorrect	25	25	25	25
Kappa Statistic	0.4182	0.4182	0.4182	0.4182
MAE	0.3237	0.3302	0.2824	0.3304
RMSE	0.4477	0.4599	0.4948	0.4577
TP Rate	0.6	0.6	0.6	0.6
FP Rate	0.182	0.182	0.182	0.182
TN Rate	0.818	0.818	0.818	0.818
FN Rate	0.4	0.4	0.4	0.4
Precision	0.75	0.75	0.75	0.75
Recall	0.75	0.75	0.75	0.75
F-Measure	0.75	0.75	0.75	0.75
ROC Area	0.677	0.668	0.623	0.674

that are correctly predicted as important. The false positive rate, FP, are the instances that are incorrectly predicted as important. The true negative rate, TN, are the instances that are correctly predicted as not important. The false negative rate, FN, are the instances that are incorrectly predicted as not important. Precision in Table 4.2 and Table 4.3 is the weighted average of the precision of the yes and no classifications. Precision for a classification is calculated as the true positives divided by the true positives plus the false positives. Recall for a classification is calculated as the true positives divided by the true positives plus the false negatives. The F-measure is the harmonic mean of precision and recall, meaning that it is calculated as twice the precision multiplied by recall and divided by the precision plus the recall. The ROC area is the area under the curve of the receiving operating characteristic curve. The curve is created by plotting the true positive rate against the false positive rate of a classification. The following sections will discuss the classification algorithms that were used and why they were considered for this problem.

Table 4.3: Classification Comparison No Discretization

Evaluation	J48	MP	NB	IBk
Percent Correct	79.1667	79.1667	79.1667	75
Percent Incorrect	20.8333	20.8333	20.8333	25
Kappa Statistic	0.4771	0.4771	0.4558	0.4182
MAE	0.2573	0.242	0.2082	0.2615
RMSE	0.4187	0.4116	0.4561	0.4902
TP Rate	0.533	0.533	0.467	0.6
FP Rate	0.091	0.091	0.061	0.182
TN Rate	0.909	0.909	0.939	0.818
FN Rate	0.467	0.467	0.533	0.4
Precision	0.785	0.785	0.785	0.75
Recall	0.792	0.792	0.792	0.75
F-Measure	0.782	0.782	0.583	0.75
ROC Area	0.754	0.802	0.774	0.667

J48

J48 is a classification algorithm that creates a C4.5 decision tree, [90]. The C4.5 decision tree works by first checking for any existing base cases. These base cases include: all instances belonging to the same class, no attributes have information gain, and an instance of a class that the algorithm has not seen yet. If none of these base cases are occurring, the normalized information gain for each attribute is calculated if the tree were to split on that attribute. The attribute that gives the highest normalized information gain is picked for the next leaf in the decision tree. This leaf servers as the next decision node in the tree and a binary split on the node occurs. The subset of attributes that is created by splitting on the best attribute is recursively traversed repeated the previous processes until a base case is reached. Once the algorithm is finished, a tree is created starting with the nodes with the highest normalized information gain near the top and those with the lowest at the bottom. WEKA allows for the user to set the minimum number of nodes in the tree and to determine whether or not the tree should be pruned to reduce the total number of nodes. This algorithm was chosen because it utilizes the normalized information gain and does not require as much setup as some other classifiers. Since this is exploratory research, it is best to pick simpler algorithms at the start to get an understanding of the data and then move on to more complex testing methods later.

Multilayer Perceptron

The multilayer perceptron is a specific type of artificial neural network called a feed-forward network. This means that the flow of data only goes from the input layer to the output layer and does not contain any recurrence in the data flow. Since there are no loops in the network, it is easier to setup and to manipulate to see the effect on the predictions. This keeps with our desire to start with simpler classifiers first to see what kind of algorithm works best with the metrics in the data set and then to utilize more detailed and complex algorithms later, if necessary. The multilayer perceptron utilizes a technique known as back-propagation to train the nodes in the network when used with an optimization method for the data. Essentially, each of the nodes in the network has a weight that it applies to whatever value gets sent to it. There is an input layer, some number of hidden layers and an output layer. In this experiment, back-propagation and gradient descent are used to calculate the ideal weights for each node. This means having the input data run through the network over and over with each iteration changing the weights in the nodes until each node reaches an optimum value that yields the best possible prediction rate. Each of the nodes utilizes a sigmoid function with the input value and the weight to calculate the output to send to the next node. A sigmoid function is one that is always between negative one and positive one. The results of this classifier consistently created an identical confusion matrix to the J48 classifier, as is evident in the results from Table 3.

Naive Bayes

The Naive Bayes classification algorithm, [91], utilizes estimator classes based on precision values that are calculated based on the training data. For each classification, analytics are found for each of the attributes: mean, standard deviation, weighted sum, and precision. The key aspect to using Naive Bayes is the assumption that the attributes are unrelated to each other and that the presence of one attribute does not impact any other attributes in the data set. Naive Bayes was chosen because it generally needs fewer data points in order to accurately estimate the data analytics of each of the attributes. The assumption that all the attributes are independent means that it is not necessary to calculate covariances between the attributes, and only the variance within each attribute is calculated. It is important to note that if the attributes are not truly independent, the results of using Naive Bayes are unlikely to be a a good choice for a classification algorithm. This classification algorithm is straight-forward, but still useful for dealing with complex systems. The classifier works best when used in supervised learning instances in order to predict classes where the class information of the training data is included. Even though recent research has shown the algorithm to be outperformed by newer prediction approaches, we chose this classifier due to its simplicity of design. For the purpose of this study, this is the worst performing classifier in the first experiment.

K-Nearest Neighbor

The last classifier tested is a version of the K-nearest neighbor classifier, IBk [92], that expands on the nearest neighbor algorithm to reduce the high storage requirements. The training data is placed on a multidimensional space and a user defined constant, k, is used along with a test point in the space. This test point is given the most common classification between the k nearest points to itself. A distance metric is used to determine which data points are closest to the test point for the surveying process. A major drawback of this classification algorithm is that when one classification is significantly more common than another, it is more likely that the test data will be skewed towards the more popular classification. Weights can be applied to the nearest nodes to the test point to give less popular nodes an advantage on the distance calculated for determining which k nodes to included

Table 4.4: Cost-Sensitive Analysis (5x/10x)

Evaluation	J48	MP	NB	IBk
Percent Correct	68.75	70.8333/68.75	79.1667	75
Percent Incorrect	31.25	29.1667/31.25	20.8333	25
Kappa Statistic	0.3333/0.375	0.4074/0.3939	0.4558	0.4182
TP Rate	0.667/0.8	0.8/0.867	0.467	0.6
FP Rate	0.303/0.364	0.333/0.394	0.061	0.182
TN Rate	0.697/0.636	0.667/0.606	0.939	0.818
FN Rate	0.333/0.2	0.2/0.133	0.533	0.4

in the survey. These weights work similarly to those in the neural network, where the input value would be the distance and the weight is multiplied by this distance. An important consideration when using the k-nearest neighbor algorithm is the value of k itself. A higher k means more nodes are consulted about the value to assign to each data point, so there is less influence from more popular training points and a more accurate assignment can be made. The drawback is that it becomes harder to create clear groupings for the classifications in the space. This algorithm was chosen for a similar reason to the others, it is easier to understand and allows for easy customizations in order to get some initial data about the trends in the data set. Another advantage to using this algorithm is that it is conducive to working as a weighted classifier, which will become important when doing cost-sensitive analysis.

4.1.6 Cost-Sensitive Analysis

A major consideration to be made when doing any type of study on evaluating data is the context of where the data comes from, and what the results mean. In this study, the purpose is to learn the trends particular to open-source software metrics and the impact of classic metrics on determining important functions as opposed to the relative metrics defined in the previous work, [1]. Remember that the main purpose for this research is to make the process for identifying key functions in a software system more efficient, not necessarily to get the best overall prediction rating possible. Ideally a predictive model would locate all of the important modules, while eliminating all of the unimportant modules. This is unrealistic, but it is possible to weight the classifiers in such as way that gives a higher cost to false negatives over false positives that would allow for more ”‘yes”’ classifiers to be correctly predicted at the cost of more incorrect ”‘no”’ predictions. This is an acceptable trade-off because the cost for having incorrect ”‘no”’ predictions is additional testing, but having a small percentage of unnecessary tests is a reasonable price to pay for more reducing the number of false negatives. Table 4.4 shows a repeat of the tests from Table 4.3, except that a cost matrix is included is constructed so that it is worse to get false negatives than false positives in our classification. The CostSensitiveClassifier is a meta classifier that can be used in WEKA with a base class to make them cost-sensitive. The process is done in this case by reweighing the training instances according to the total cost assigned to each class represented by a 2 x 2 cost matrix.

The cost-sensitive analysis was repeated twice, once with false negatives being five times the cost of false positives and again with false negatives being ten times the cost of false positives. This is represented in the table in the form x/y where applicable, meaning that x is the five times cost and y is the ten times cost if there was any change in that value with the increase in cost. Table 4.4 contains the data from both of these experiments where applicable. It should be noted that even by increasing the cost of false negatives by ten, the results of the experiment were the same for Naive Bayes and for IBk. On the other hand, J48 and particularly the Multilayer Perceptron show good increases in true positive rate.

4.1.7 Discussion

The initial results found in these tests would indicate that machine learning is a strong choice for increasing the efficiency of identifying the key functions in a software system for completing reliability verification testing. The results suggest that the relative metrics defined according to the intended use cases have a greater impact on the key locations in a system for placing error handling. Standard metrics, such as McCabe and Halstead, are less effective in determining these key areas once a relative portion of the code is being examined. For the purposes of the research, it would seem that focusing on the real-world impact of faults in a system and the metrics which put that into perspective are of greater use than just code or design metrics alone. It was shown that discretization yielded a consistently higher true positive rate while reducing total prediction accuracy in most cases. Without cost-sensitive analysis it makes sense to utilize supervised discretion to pre-process the data and allow for a more effective prediction model which is not heavily impacted by the choice in classification algorithm. But, the overall best results for our purposes came from the cost-sensitive analysis. Despite having a lower total prediction rate, J48 and the Multilayer Perceptron classifiers made substantial increases to the true positive rate. As stated previously, it is far more important to us to have as high a true positive rate as possible as opposed to a high total percent correct.

4.2 Variable Metric Predictions

4.2.1 Case Study

The case study described in this section is based on the open-source software project same as the previous experiment. The difference between the last experiment and this one is that the previous data acquired by testing and analyzing the Mp3gain project was repeated using the updated framework and the new metrics were used for predicting the variable priorities. The goal of this study was to re-test Mp3gain and extract the software metrics as features for a machine learning algorithm to use to predict the relative importance of variables. Using this relative importance the framework can rank the variables according to their risk to the system with respect to the reliability of the system when under duress from data faults occurring in memory. By utilizing a combination of widely accepted metrics and custom metrics, the machine learning algorithms are used to identify trends and help to determine which metrics provide the greatest amount of information for predicting the relative importance of the variables in the system. The predictions made by the machine learning algorithm are verified by using the fault injection data to calculate the revised relative importance value to use against the predictions to determine the accuracy of the model.

4.2.2 Test Setup

The test setup repeats the test case from [1] with the improvements made to the fault injection framework discussed in the previous chapter. The testing was done using a wide range of available attribute selection and classification algorithms. The algorithms that produced the most relevant results are included in the experiment. Since the machine learning component of the framework is the newest aspect of the research and given there is no available literature pertaining to how machine learning could be used for the specific purposes of this research, a wide survey was completed in order to narrow down what types of algorithms would be most useful for the framework. The WEKA program is used and ten-fold cross-validation is used in order to get more accurate data from the datasets. The sample size is still relatively small at this stage of the research and ten-fold cross-validation is the best way to get relevant results about the potential of the predictive model used in the framework.

4.2.3 Attribute Selection

To identify which attributes have the greatest merit when predicting the value of relative importance for each of the variables in the dataset, two attribute selection algorithms are used. The full training set is used as an input to the attribute selection algorithms. In this case, not all of the algorithms worked utilizing ten-fold cross-validation and so the entire training set is used in order to maintain consistency between the algorithms. First, the metrics that were used as features in the dataset are discussed in detail.

Metrics Used

Function Metrics All of the metric values from the function pertaining to each individual variable are included in the dataset. At this point it was important to include as many features as possible in order to identify any possible trends between the software metrics in order to evaluate which metrics were the best for predicting the relative importance. Refer to the previous experiment for details about these metrics. The only new metric to add to this section is the inclusion of the P_{Cf} that was added with the improvements to the metrics and was calculated for each function in the target software and used to recalculate the priority of each function accordingly.

Static Metrics The new static metrics included in the framework analysis are: local connection, LC, intermodule connections, IMC, reads, writes, variable type. The local connections are the number of variables that effect or are effected by the variable in question. This was determined using the source code of the target software and manually investigating each line of code that the variable occurred in. This value can be determined during the dynamic code slicing stage of the framework. The intermodule connections refers to the number of the functions that the variable appears to effect. This includes all function calls within the function currently being investigated. All variables are given a starting value of one local and intermodule connection and for each new connection that value is increased by one. Each branch statement that relies on the variable value increases the local connections by one. In the case of global variables, the intermodule connections value increases by one. The reads and writes are the number of times in the function that the value of the variable is read from and written to. This is done because it is important for helping to predict the temporal impact and potentially the failure rate. Each write to the memory location for a variable is one potential overwrite of the faulty data. Each read is potentially a point where the data fault can affect the system. The variable type has proven to be an interesting metric to include in the datasets. The different variable types have a widely varying effect on the system when injected with faults. For example, with the implementation of bool variables the failure rate is particularly low because of the chances of flipping the single bit used to determine value. The failure rate for string variables tends to be higher than other variable types, especially with variables dealing with file paths and file names.

Dynamic Metrics The new dynamic metrics include the frequency and size. The frequency of the variable, VarFreq, is calculated by using the results of the gcov tool. Each occurrence of the variable in the function currently being analyzed and the frequency of the lines of code that are executed with the variable in them are summed to give the frequency that the variable occurs throughout the execution of the program. This value serves as the variable equivalent to the frequency found for each function call using the gcov tool to determine the probability of execution of each function. The size metric is calculated by instrumenting the code with a probe that returns the size of the variable in bytes. This is done because the size of variables is implementation specific and it is important to identify the size of the variables for the particular system being used. For this reason both the size and the variable type are included in the dataset. The size is used in order to calculate the probability of corruption for the specific variable.

Relative Metrics The primary relative metrics included in the dataset are P_{Ev}, P_{Cv}, P_{FI}, and relative importance. Just as with the function metrics, the probability of the randomly generated data fault occurring in memory belonging to the variable and then being accessed by the program are determined in order to add an element of relevance to the importance calculation. The probability of these two independent events occurring at the same time is the P_{FI} metric. This is the value that is multiplied by the importance of the variable to calculate the relative importance. Refer to the previous chapter for a more detailed look at how these values are calculated.

Merit and Rank of Attributes

Principal Component with Ranker The principal component algorithm performs an analysis and transformation of the data. It is meant to be used in conjunction with a Ranker search to order the attributes according to relevance. Dimensionality reduction is accomplished by choosing enough eigenvectors to account for some percentage of the variance in the original data, in this case a variance of ninety-five percent is used. Attribute noise is filtered by transforming to the PC space, eliminating some of the worst eigenvectors, and then transforming back to the original space. Dimensionality reduction in this context is the process of reducing the number of random variables under consideration for feature selection. Table 4.5 shows the coefficients for each attribute determined through the principal component method. The data in the table refers to the highest ranked attribute eigenvector from the output of the algorithm. Essentially, each value of the attribute would be multiplied by its coefficient and added to the other attributes in order to obtain the class value.

Relief Attribute Evaluation The relief attribute evaluation algorithm evaluates the worth of an attribute by repeatedly sampling an instance and considering the value of the given attribute for the nearest instance of the same and different class. This algorithm detects the features which are statistically relevant to the desired class. Table 4.6 shows the rankings of the attributes according to this algorithm. The influence of each attribute indicates how much of an effect the attribute has on the prediction of the relative importance. Higher numbers indicate a greater influence on the prediction and the positive and negative values indicate a direct or indirect proportion to the class, respectively. More information can be found at [93].

Table 4.5: Mp3gain Principal Component

Coefficient	Attribute	Coefficient	Attribute
-0.0215	vartype=real	0.2413	N1
-0.0337	vartype=real*	0.2363	N2
-0.0385	vartype=int	0.2416	Vocab
-0.0027	vartype=int*	0.24	Length
0.0008	vartype=Float_t*	0.2409	Volume
0.057	vartype=uchar	0.1468	Difficulty
0.0552	vartype=uchar*	0.2409	Effort
0.0035	vartype=ulong	0.2373	Mental
0.0773	vartype=long	0.241	ModLoC
-0.0055	vartype=uint	0.0991	Branches
-0.0047	VarFreq	-0.0629	Comments
0.0472	LC	0.241	LogLoc
0.0132	Reads	0.238	Complexity
-0.0018	Writes	0.2282	MaxNested
-0.0126	IMC	0.2243	AvgNested
-0.0047	PoEv	-0.0704	FuncFreq
-0.0556	size	0.2254	propagation
-0.0556	PoCv	-0.0118	ELoC
-0.004	PoFI	-0.0118	PoEf
0.1909	n1	0.2368	PoCf
0.2416	n2	0.2368	Mem

4.2.4 Classification

ZeroR

This is the default algorithm for classifying the dataset. It serves as the starting point for validating the performance of other classification algorithms on a dataset. This classification has no rules, so it merely predicts the mean, for numeric classes, or the mode, for nominal classes. This algorithm is included in order to give a reference point to the predictions made by other algorithms. Algorithms that do worse than ZeroR are worse than just predicting the mean each time. For the purposes of this framework, ZeroR and the majority of the rule based algorithms are not of much use because they create a series of rules that provide threshold values for the predictions. The algorithms just predict whichever threshold value

Table 4.6: Mp3Gain Relief Attribute Evaluation

Influence	Attribute	Influence	Attribute
0.747	PoFI	-0.0195	Mental
0.7413	VarFreq	-0.0204	Vocab
0.7413	PoEv	-0.0209	n1
0.6993	Reads	-0.0213	n2
0.2188	Writes	-0.0216	Complexity
0.2151	vartype	-0.0224	LogLoc
0.2005	LC	-0.0226	ModLoC
0.1755	size	-0.0229	Mem
0.1755	PoCv	-0.0229	PoCf
-0.0138	PoEf	-0.0274	AvgNested
-0.0138	ELoC	-0.0291	MaxNested
-0.0149	Difficulty	-0.0308	propagation
-0.0189	N1	-0.0377	FuncFreq
-0.0191	Length	-0.0405	Branches
-0.0193	Effort	-0.045	IMC
-0.0193	Volume	-0.055	Comments
-0.0194	N2		

the dataset prediction comes out closest to. Since the framework seeks to rank the output of these algorithms in order to identify the top ten, twenty and so on percent of variables, this type of prediction does not work for this purpose.

Linear Regression

A regression algorithm is included as another reference point to compare the other machine learning algorithms against. This is a simple classifier that uses linear regression for prediction. The algorithm uses the Akaike criterion for model selection, and is able to deal with weighted instances. The attribute selection is done by stepping through the attributes, removing the one with the smallest standardized coefficient until no improvement is observed in the estimate of the error given by the Akaike information criterion. The Akiake criterion is as follows, where L is the maximized value of the likelihood function for the model and k is the number of estimated parameters in the model. This means that given a set of candidate models for the dataset, the best model is determined based on which model has the lowers AIC value. The linear regression model fits a straight line through the set of data points in order to make the sum of the squared values for the distance between the points of the data set as small as is possible. The resulting model is used to predict the desired class.

$$AIC = 2 * k - 2 * ln(L) \tag{4.2}$$

IBk

This is an implementation of the K-nearest neighbor classifier. [92] See the previous experiment for a brief description of its functionality.

KStar

K* is an instance-based classifier, that is the class of a test instance based upon the class of those training instances similar to it, as determined by some similarity function. It differs from other instance-based learners in that it uses an entropy-based distance function. Using the entropy as a distance measure provides this algorithm with benefits over other: consistent handling of symbolic attributes, real values and missing values. Instance based learners work by comparing the current instance, or data point, to a database of pre-classified examples, the training set. This explains the poor performance of this variable in the validation testing that tested the models using new unseen test data instead of cross-validation. Since the dataset is relatively small, it is unlikely that there will be many instances for the algorithm to compared the instances to in order to get an accurate prediction. More information is available from [94]

LWL

The LWL algorithm stands for locally weighted learning. This algorithm uses an instance-based algorithm to assign instance weights which are then used by a specified

WeightedInstancesHandler method in WEKA. This algorithm then utilizes another algorithm depending on the desired prediction values. For classification predictions Naive Bayes is used and linear regression is used for regression models, like this one. This model, like the previous two, are considered lazy model types, in that they are generally good at predicting values for data point types that they have seen before. The results show that the lazy types of predictors do well with the cross-validation testing because the dataset is balanced between training and test data. The exception to this is the validation testing using just the mv variable dataset that the predictor has not seen before. This problem is potentially solved by adding more data points over varying types to the dataset to give the algorithm a wider range of data points for comparing the test data. More information on the functionality of the algorithm can be found at [95].

Least Median Squared

This algorithm implements a least median squared linear regression algorithm that utilizes the existing linear regression class in WEKA to form predictions. Least squared regression functions are generated from random subsamples of the data. The least squared regression with the lowest median squared error is chosen as the final model. This model is an interesting addition to the experimental data, because even though it is based around the linear regression model that provides less desirable prediction rates, this algorithm is consistently a strong predictor. Particularly in dealing with the test data from mv that the model has not seen before in the next section. The model minimizes the sum of the squared residuals, the same as linear regression. For this specific algorithm, the square is replaced by the median o the squared residuals. More information about this algorithm can be found at [96] and [97].

M5P

This algorithm implements base routines for generating an M5 model and trees. The original algorithm M5 is defined in [98] and improvements are described in [99]. The algorithm deals with inducing trees of regression models. The algorithm works by creating a decision-tree induction algorithm to build a tree where each node minimizes the variation of the class values for the nodes beneath it on its branch. The next step prunes the tree from each individual leaf using the original M5 algorithm. The primary difference between the old algorithm and this one is that when pruning an interior node the algorithm decides between replacing the node with a constant value or a regression plane. The attributes of the regression plane are those that are used in decisions in the tree nodes that are located lower in the branch than the current node. [99]

4.2.5 Discussion

The results of the principal component attribute selection indicate that the attributes of P_{FI}, $ELoC$, P_{Ef}, writes, P_{Ev}, and IMC have the greatest natural correlation with the predicting of the relative importance class. In the case of the principal component, the smaller the coefficient, the less the attribute has to be adjusted in order to predict the class. This means that coefficients close to zero are the most indicative of a strong relationship with the class. The attribute merit of the relief attribute evaluation algorithm has P_{FI}, variable frequency, P_{Ev}, and reads as the highest ranked attributes for predicting the relative importance. The inclusion of variable frequency and P_{Ev} with the same influence makes sense, since the frequency is used when calculating P_{Ev}. In both cases the P_{FI} value is used, which makes sense because it is directly proportional to calculating the relative importance, but it is interesting to see the non relative metrics, like reads, writes, variable type, LC, size and variable frequency having a notable influence on the prediction of relative importance. This indicates that there is potential in predicting the relative importance using both the relative metrics and more conventional code metrics.

Table 4.7: Mp3gain Variable Evaluation

Model	Total Mean Error	Mean Absolute Error	Root Mean Squared Error
zeroR	2.90	32.89	381.48
Linear Regression	1.11	25.59	307.78
Ibk	0.40	12.23	164.86
Kstar	2.14	27.31	289.33
LWL	1.86	21.27	252.14
LeastMedSq	0.20	5.63	83.27
M5P	0.81	21.95	264.44

The evaluation metrics collected from the prediction testing of the Mp3gain dataset are detailed over the next three tables. Table 4.7 shows the total mean error , mean absolute error and root mean squared error for each of the models. The total mean error is the averages of the difference between the predicted relative importance rank and the actual relative importance rank of the variable. The mean absolute error is the sum of the differences in the actual and predicted ranks and divided by the total number of data points. It is a measure of the average difference in the ranking potential of the model across all data points. The root mean squared error is found by taking the square root of the sum of each difference in the rankings squared. This value only has worth when comparing models that work with the same data types and context, so they serve as another means of measuring the models used in this dissertation to one another. It is important to note here that framework is not interested in getting the predicted ranks to be exactly identical to the actual ranks, but more in the trend of the ranking process. Even if the overall rankings are off by a few ranks, as long as the majority of the ranks fall into the correct range, top ten percent, twenty percent and so on, then these models have achieved a useful result.

It can be seen in Table 4.7 that for ten-fold cross validation on the revise Mp3gain dataset that the total mean errors indicate a strong result from IBk and the LeastMedSq algorithms. The model selection is a balance of functional, lazy, rule based and tree type algorithms.

Table 4.8: Mp3gain Variable Average Percent Correct

Percentage	zeroR	Linear Regression	Ibk	Kstar	LWL	LeastMedSq	M5P
10	0.00	0.60	0.40	0.10	0.20	0.80	0.60
20	0.05	0.42	0.68	0.32	0.47	0.79	0.58
30	0.21	0.50	0.86	0.39	0.64	0.96	0.68
40	0.38	0.59	0.89	0.46	0.70	0.95	0.70
50	0.46	0.59	0.83	0.41	0.67	0.93	0.63
60	0.56	0.64	0.80	1.00	0.67	0.93	0.69
70	0.72	0.66	0.83	1.00	0.77	0.92	0.72
80	0.81	0.78	0.93	1.00	0.84	0.95	0.79
90	0.90	0.89	0.93	1.00	0.91	0.96	0.89
100	1.00	1.00	1.00	1.00	1.00	1.00	1.00

The value in Table 4.8 show the average percent of the data points ranked within the correct ranking range. Here it can be seen that all of the models eventually work their way up to being particularly accurate in identifying the top eighty or so percent of variables, but the framework is mostly interested in the results in the top thirty percent. The ranking accuracy tends to be a bit lower in the top ten percent, but tends to increase to a better prediction rate at twenty percent. An important note here for algorithms like zeroR and KStar is that they will utilize threshold values and predict that all of the data points fall on one of those threshold values instead of predicting an exact value for each point. In particular, the KStar algorithm predicts any numbers near zero as zero, so for all the data points beyond rank fifty-five it predicts zero. This isn't necessarily a bad thing, as it is unlikely that the target software being testing by the framework would be capable of maintaining its standard operation after adding error handling to fifty or more variables. Of note here is the high percent correct of LeastMedSq and IBk. This will be a recurring trend in the remaining data, and indicates that these two algorithms are best to pursue in future research. Figure 4.1 gives a visual representation of the individual models and their accuracy in predicting variables in the various percent ranges of the rankings.

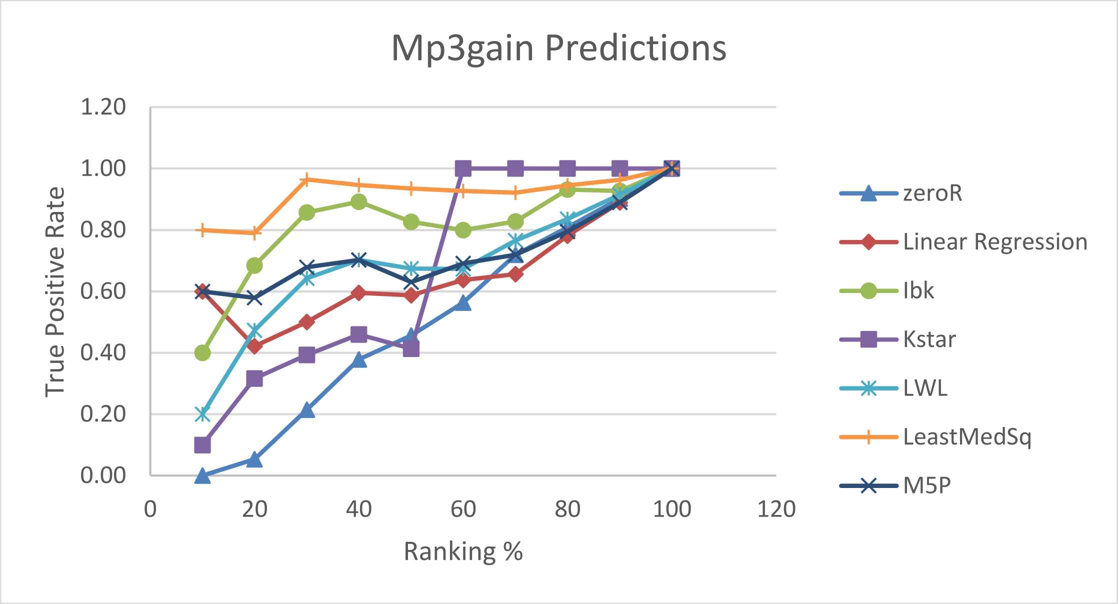

Figure 4.1: Mp3gain Predictions

In order to take another look at the variance in the predictions between the models, Table 4.9 shows the average ranking difference found across all the predictions for each of the models. The way to interpret the model is that the values indicate the average number of rankings that the model is off, either by predicting too high or too low. For example, that means when predicting the actual top ten percent ranked variables, the average error in the actual ranking predictions is 2.60 for LeastMeanSq. Since the framework is less concerned with predicting the exact value and more concerned with predicting the trends, this value is not as important as the previous one, but is a useful comparison metric. The table shows that there is a substantial error in guessing the rankings of variables in models using anything other than LeastMeanSq, IBk, and M5P. Note that even with a high variance in correctly identifying the actual rank, the overall trends of the predictive models are close enough to obtain high percent correct, near seventy percent or higher, despite this variance. Figure 4.2 shows the relationship of the different models and their average rank difference across each ranking range.

Table 4.9: Mp3gain Variable Average Rank Difference

Percentage	zeroR	Linear Regression	Ibk	Kstar	LWL	LeastMedsq	M5P
10	57.70	25.90	6.70	44.80	31.10	2.60	11.90
20	45.21	25.68	6.95	34.42	25.74	4.21	18.37
30	38.21	24.68	7.82	30.64	24.39	4.82	17.39
40	34.00	24.03	8.19	28.32	22.05	4.35	18.38
50	32.24	24.39	11.00	25.37	22.07	4.93	20.57
60	30.49	24.38	11.75	24.55	21.36	4.96	20.27
70	30.88	23.16	11.81	25.20	21.11	5.69	20.33
80	30.37	23.07	11.70	25.36	21.58	5.37	21.08
90	30.89	23.72	11.54	26.32	21.21	5.50	20.65
100	32.89	25.59	12.23	27.31	21.27	5.63	21.95

4.3 Validation Test

4.3.1 Case Study

The case study for this experiment was to complete the entirety of the framework process on a new dataset that had not been seen by the models and then to combine the previous dataset and the new dataset for one larger dataset and to observe the improvements with an increase in data points. The dataset used in this experiment is taken from the testing and analysis of the Mv program from the coreutils. The Mv tests come from testing the models by using the Mp3gain datasets as training data and the Mv datasets as test data. The second part of the validation testing was to combine these models and complete ten-fold cross validation in order to get a more accurate reading of the prediction potential of the models once more data points are collected.

4.3.2 Test Setup

The test setup for each of the sections, function priority and variable priority, are the same as the two previous sections of the experimental design chapter. The only difference is that the datasets used came from running the framework on new software. The validation testing was done in the exact same way as the tests for creating and testing the baseline

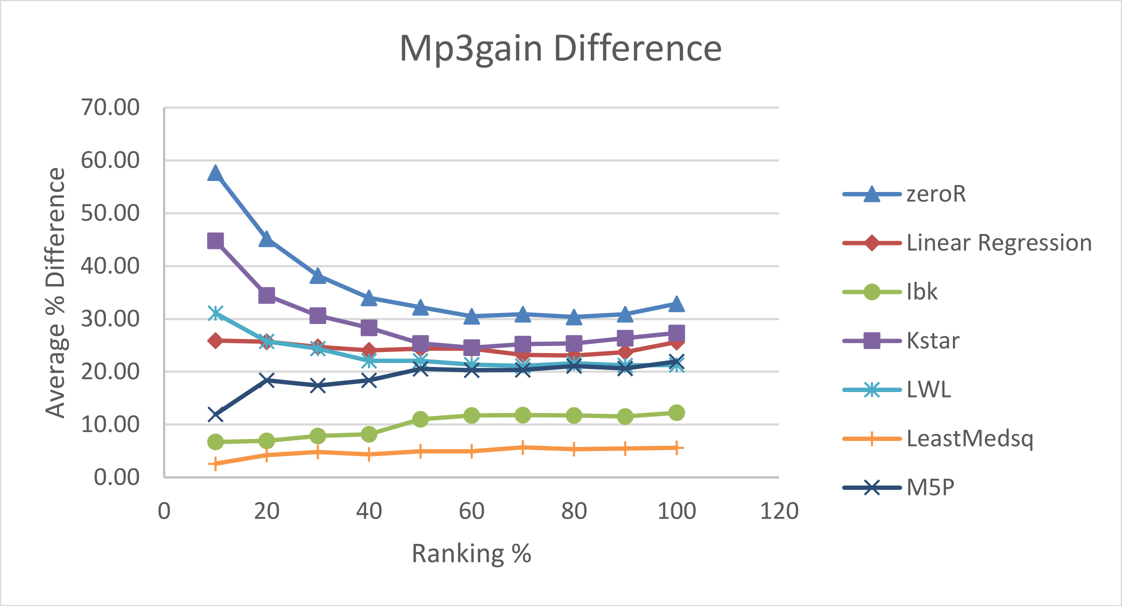

Figure 4.2: Mp3gain Average Percent Difference

models for predicting function priority and variable priority. The rest of this section will discuss the evaluation of the attributes and predictions for the function prediction model and then the variable prediction model. The data was collected during the execution of the framework on the Mv program and once the framework had completed fault injection analysis and calculated the relative importance, the datasets were run through WEKA to investigate the capability of the predictive models to predict on data that had not been seen before and then again using a method more relevant to smaller datasets.

4.3.3 Function Attribute Selection

The attributes selected for the dataset in this experiment are the same as the baseline function prediction experiment in the previous section. The exception is the inclusion of the P_{Cf} metric which is used to calculate the priority of the functions.

Table 4.10: Validation Function Attributes Info Gain

average merit	average rank	attribute
0.324 +- 0.03	1.4 +- 0.8	Propagation
0.18 +- 0.119	4.2 +- 3.6	Difficulty
0.053 +- 0.107	5.5 +- 0.67	Volume
0.053 +- 0.107	5.8 +- 2.27	Length
0 +- 0	6.8 +- 4.4	Mem
0.025 +- 0.075	8.2 +- 2.4	Vocab
0.151 +- 0.128	8.9 +- 6.67	ELoC
0.053 +- 0.107	9.3 +- 3.61	Effort
0.151 +- 0.128	9.5 +- 6.56	PoEf
0.053 +- 0.107	9.6 +- 0.92	N1
0.053 +- 0.107	10 +- 1.1	N2
0.053 +- 0.107	10.4 +- 3.23	Mental
0.025 +- 0.075	10.8 +- 1.47	n2
0.035 +- 0.106	12.2 +- 5.13	ModLoc
0 +- 0	14.1 +- 1.14	Branches
0.046 +- 0.092	15.9 +- 4.7	Frequency
0.02 +- 0.06	17.6 +- 1.91	MaxNested
0.023 +- 0.069	17.8 +- 1.4	AvgNested
0.029 +- 0.088	17.9 +- 4.85	Complexity
0 +- 0	17.9 +- 2.84	PoCf
0.035 +- 0.106	19.1 +- 6.09	LogLoc
0 +- 0	20.1 +- 2.21	Comments
0 +- 0	23 +- 0	n1

4.3.4 Merit and Rank of Function Attributes

Information Gain Attribute Evaluation with Ranker

The information gain attribute evaluation algorithm is detailed in section 4.1.3.

Correlation Attribute Evaluation with Ranker

This algorithm evaluates the worth of an attribute by measuring Pearson's correlation between it and the class. Nominal attributes are considered on a value by value basis by treating each value as an indicator. An overall correlation for a nominal attribute is arrived at via a weighted average. Pearson's correlation in this context is the measure of the linear

Table 4.11: Validation Function Attribute Correlation

average merit	average rank	attribute
0.622 +- 0.033	1.3 +- 0.9	Propagation
0.537 +− 0.044	3 +- 0.77	N2
0.551 +- 0.06	3.2 +- 3.63	Difficulty
0.518 +- 0.043	4.6 +- 1.2	Length
0.504 +- 0.052	6.3 +- 2.69	Vocab
0.503 +- 0.044	6.4 +- 1.5	N1
0.497 +- 0.039	7.7 +- 1	Effort
0.495 +- 0.051	7.9 +- 2.84	n2
0.496 +- 0.039	8.8 +- 1.17	Volume
0.478 +- 0.045	11 +- 2.68	Mental
0.474 +- 0.056	11.2 +- 1.6	ModLoc
0.462 +- 0.061	11.9 +- 3.83	MaxNested
0.465 +- 0.059	12.7 +- 2.37	LogLoc
0.448 +- 0.042	14.2 +- 1.17	n1
0.438 +- 0.065	14.7 +- 2.37	Complexity
0.432 +- 0.051	16 +- 1.18	AvgNested
0.394 +- 0.093	17 +- 1.61	Mem
0.394 +- 0.093	18 +- 1.61	PoCf
0.347 +- 0.117	18.2 +- 4.24	PoEf
0.347 +- 0.117	18.6 +- 4.59	ELoC
0.237 +- 0.097	20 +- 5.37	Frequency
0.251 +- 0.067	20.8 +- 1.25	Comments
0.169 +- 0.062	22.5 +- 1.02	Branches

correlation, dependence, between each attribute and the class value. The resulting merit value is between -1 and 1, where a positive value indicates that a linear equation describes the relationship between the attribute and the class and a negative value indicates that the data points lie on a line where when the class value decreases, the attribute value increases.

Gain Ratio Attribute Evaluation with Ranker

The gain ratio attribute evaluation algorithm determines the worth of an attribute by measuring the gain ratio with respect to the class using the following equation, where H is the information entropy as discussed in the previous section for the information gain attribute

Table 4.12: Validation Function Attributes Gain Ratio

average merit	average rank	attribute
0.355 +- 0.242	3.1 +- 3.48	Difficulty
0.404 +- 0.039	3.2 +- 3.46	Propagation
0.093 +- 0.189	5 +- 0.89	Volume
0.093 +- 0.189	5.4 +- 2.62	Length
0 +- 0	6.8 +- 4.4	Mem
0.028 +- 0.085	8.1 +- 2.21	Vocab
0.189 +- 0.172	8.9 +- 6.67	ELoC
0.093 +- 0.189	9 +- 1.41	N1
0.093 +- 0.189	9.1 +- 3.75	Effort
0.189 +- 0.172	9.3 +- 6.57	PoEf
0.093 +- 0.189	9.4 +- 2.01	N2
0.093 +- 0.189	10.1 +- 3.75	Mental
0.028 +- 0.085	10.9 +- 1.58	n2
0.049 +- 0.147	12.9 +- 4.06	ModLoc
0 +- 0	14.1 +- 1.14	Branches
0.06 +- 0.119	15.9 +- 4.7	Frequency
0.021 +- 0.063	17.6 +- 1.91	MaxNested
0.025 +- 0.076	17.8 +- 1.4	AvgNested
0 +- 0	17.9 +- 2.84	PoCf
0.036 +- 0.109	18.6 +- 2.91	Complexity
0.049 +- 0.147	19.8 +- 4.02	LogLoc
0 +- 0	20.1 +- 2.21	Comments
0 +- 0	23 +- 0	n1

evaluator. One advantage to utilizing this algorithm is that the algorithm creates a biased decision tree that is against attributes with a high number of distinct values. This is done in order to avoid over-fitting the data to a particular type of attribute that gives too much information about the classifier.

$$GainR(Class, Attribute) = (H(Class) - H(Class|Attribute))/H(Attribute). \tag{4.3}$$

Cfs Subset Attribute Evaluation with Greedy Stepwise

The last attribute selection algorithm considered in this validation test is the CfsSubset attribute evaluation algorithm which determines the worth of a subset of attributes by considering the individual predictive ability of each feature along with the degree of redundancy between them. The subsets of features that are highly correlated with the class while having low inter-correlation are preferred. The CFS algorithm is a feature selection based algorithm that uses a search algorithm with a particular function to evaluate the merit of various subsets of metrics. In the case of this research we will consider the highest merit subset for each of the cross fold validation tests of the algorithm. CFS measures the merit of the feature subsets based on the concept that good subsets contain features highly correlated with the class, while remaining uncorrelated with each other. More information can be found at [100]

This type of algorithm does not work with the ranker algorithm in WEKA to order the predictions according to their ranked merit, because the algorithm does not calculate the merit in the same way. Instead a greedy stepwise algorithm is used which performs a greedy forward or backward search through the space of attribute subsets. The algorithm can start with none or all of the attributes or from an arbitrary point in the space. It stops when the addition or deletion of any remaining attributes results in a decrease in evaluation. The algorithm can also produce a ranked list of attributes by traversing the space from one side to the other and recording the order that attributes are selected.

4.3.5 Function Classification

The classification algorithms that were used in this phase of the testing are the same as those used when creating the baseline model for predicting the priority of the functions. Refer to section 4.1.5 for more information.

Table 4.13: Validation Function Attribute Cfs Subset Evaluation

number of folds (%) attribute	attribute
0 (0%)	n1
1 (10%)	n2
1 (10%)	N1
0 (0%)	N2
0 (0%)	Vocab
0 (0%)	Length
0 (0%)	Volume
6 (60%)	Difficulty
0 (0%)	Effort
0 (0%)	Mental
1 (10%)	ModLoc
0 (0%)	Branches
0 (0%)	Comments
0 (0%)	LogLoc
0 (0%)	Complexity
0 (0%)	MaxNested
0 (0%)	AvgNested
0 (0%)	Frequency
10 (100%)	Propagation
6 (60%)	EloC
1 (10%)	PoEf
0 (0%)	PoCf
0 (0%)	Mem

4.3.6 Discussion: Function Validation

The attribute selection algorithms for the function prediction model tend to favor the propagation, difficulty, volume, and the N2 metric as the attributes with the highest merit for predicting the importance of the functions under consideration. These metrics generally are commonly accepted metrics that measure known factors about the function code. The exception is the propagation metric which is calculated based on the data flow graphs produced by Doxygen in order to determine how many other functions each function is connected to and could potentially pass a faulty variable to. The propagation metric is directly related to the connection metrics calculated for the variable dataset, so it makes sense that

Table 4.14: Mv Function Evaluation

Evaluation	ZeroR	J48	Ibk	MP	NB
Percent Correct	85.71	14.29	14.29	14.29	57.14
Percent Incorrect	14.29	85.71	85.71	85.71	42.86
Kappa Statistic	0	0	0	0	0.2222
TP Rate	1	0	0	0	0.5
FP Rate	1	0	0	0	0
TN Rate	0	1	1	1	1
FN Rate	0	1	1	1	0.5

it would rank highly. With the amount of data that is used at this point of the research, it is possible that these trend values will fluctuate as more types of software systems are evaluated.

The evaluation of the Mv dataset using the Mp3gain dataset as the training data gave the expected results in Table 4.14. With such a small dataset based on a single open-source project, it is unlikely that the model would be capable of properly predicting the importance of the functions in the target software. The predictive model does not have enough data points to predict with high certainty which functions are important or not. With the addition of new data from different types of software projects, size, architecture, etc. the prediction percentage will increase, as is shown later. Most of the models had a particularly low true positive rate, even with using the cost-benefit analysis technique to skew the predictions towards as few false negative as possible. Naive Bayes is the only algorithm to manage predicting more than one or zero of the important functions correctly.

Table 4.15 shows the results when the Mv dataset is combined with the Mp3gain dataset and ten-fold cross-validation is used to measure the prediction capabilities of the models. Remember that cross-validation essentially removes a random selection of ten percent of the training data as test data and uses the rest as training data and conducts the prediction. This removal and test process is repeated ten times and the complete result is what is presented. It can be seen that the prediction rates are much better with cross-validation because the

Table 4.15: Validation Function Evaluation

Evaluation	ZeroR	J48	Ibk	MP	NB
Percent Correct	23.08	88.46	82.69	78.85	84.61
Percent Incorrect	76.92	11.54	17.3	21.15	15.38
Kappa Statistic	0	0.6929	0.4293	0.4211	0.5398
TP Rate	1	0.833	0.417	0.583	0.583
FP Rate	1	0.1	0.05	0.15	0.075
TN Rate	0	0.9	0.95	0.85	0.925
FN Rate	0	0.167	0.583	0.417	0.417

models have more relevant data points to work from when creating the prediction model. This result suggests that more data is needed from a varying number and type of software projects in order to train a model to be able to accurately predict the importance of the functions. The J48 algorithm proves to be the most effective algorithm for predicting the important algorithms with a true positive rate of 0.833. This true positive rate combined with the overall accuracy of 88.46 suggests that J48 is the ideal algorithm to use for training our predictive model.

The learning curve of the validation dataset is determined by utilizing the best algorithm, J48, and removing sections of the data to measure the accuracy of the predictions based on the amount of data in the dataset. Table 4.16 and figure 4.3 illustrate this concept. It can be seen that with an increasing amount of data to work with, the algorithm accuracy gets much higher. This is an important point because it shows that with the increase in the dataset size, the accuracy of the overall model should increase. This is further evidence that further data would allow the model to more accurately handle unknown data points.

4.3.7 Variable Attribute Selection

4.3.8 Merit and Rank of Variable Attributes

The attribute selection algorithms used are the same as those used in the baseline model testing in section 4.2.3.

Table 4.16: Validation Function Learning Curve

Percent Removed	Accuracy
90	38.2
80	52.43
70	74.17
60	72.83
50	77.67
40	79.93
30	84.63
20	84.9
10	89.97

4.3.9 Variable Classification

The classification algorithms are the same as those used in the baseline model evaluation in section 4.2.4.

4.3.10 Discussion: Variable Validation

The attribute selection algorithms: principal component and relief information gain were run on the combined dataset to obtain the evaluation metrics found in Table 4.17 and Table 4.18. It can bee see that with the additional data added to the training dataset, the principal component algorithm chose P_{Ev}, function frequency, P_{FI}, and a few of the variable types as the attributes with the highest impact on the relative importance of the variable. This makes sense when you consider how the relative importance is calculated. The P_{FI} is directly proportional to the value of the relative importance, but the other attributes help to identify some trends in the metrics that can be used to determine the relationship with the other values that make up the relative importance. The type of variable, function frequency, probability that a variable will be executed after having a fault injected and the type of variable all appear to have a closen connection to the failure rate, spatial impact and temporal impact. The relief attribute evaluation provides a similar attribute selection, identifying P_{FI}, variable frequency, P_{Ev} and the read metric as being the strongest indicators

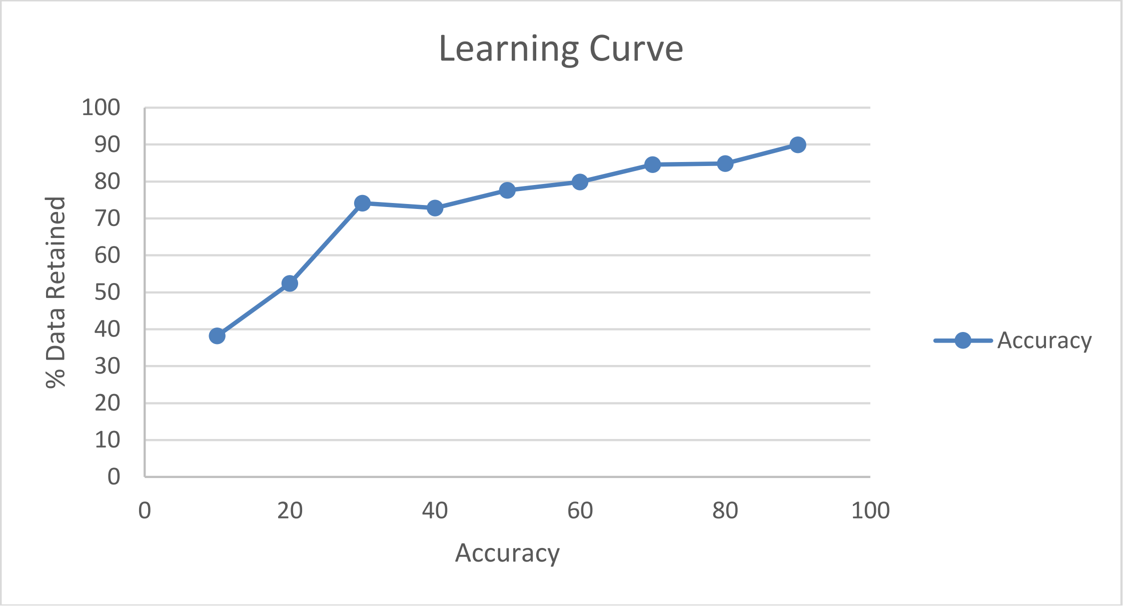

Figure 4.3: Function Learning Curve

of predicting the correct priority of the variables. Just as before, these values make sense, as the variable frequency is tied to the probability of a fault occuring during the program executing a line of code involving the particular variable and the reads helps to show the likelihood that the data fault will have an effect on the system.

The variable evaluation results are similar to those from the function evaluation when tested along on the models trained by the Mp3gain dataset. Given the dataset sizes used, it makes sense for there to be a drop in accuracy when dealing with unseen data as opposed to the validation dataset that uses the combined data from Mv and Mp3gain and cross-validation. As shown in Table 4.19 there is a drop in accuracy from the previous run of the models on just the Mp3gain datasets. The total mean error is quite high for most all cases, the exception being the least median squares algorithm. This indicates that against data that has not been seen before, with a small sample size for training, the other models predicted quite poorly when compared to the zeroR model which just predicts the mean value for each. But, when combining the datasets and using the ten-fold cross-validation method in order to test on a more knowledgeable dataset, the error found in the models shown in Table 4.20 is back in line with where it would be expected. Note again that the

Table 4.17: Validation Principal Component

Coefficient	Attribute	Coefficient	Attribute
0.0022	vartype=real	0.1984	n1
0.0065	vartype=real*	0.2387	n2
0.0317	vartype=int	0.2489	N1
0.0073	vartype=int*	0.2457	N2
0.017	vartype=Float_t*	0.2423	Vocab
0.0606	vartype=uchar	0.2482	Length
0.066	vartype=uchar*	0.2474	Volume
0.0205	vartype=ulong	0.1808	Difficulty
0.0831	vartype=long	0.2475	Effort
0.0031	vartype=char**	0.2435	Mental
-0.0326	vartype=enum	0.2474	ModLoC
-0.0801	vartype=bool	0.0394	Branches
0.0034	vartype=uint	-0.0228	Comments
-0.0323	vartype=char*	0.2443	LogLoc
0.0233	VarFreq	0.2431	Complexity
0.0765	LC	0.2267	MaxNested
0.045	Reads	0.223	AvgNested
0.0175	Writes	-0.0098	FuncFreq
-0.0021	IMC	0.2342	Propagation
0.008	PoEv	0.0388	ELoC
0.0506	size	-0.0352	PoEf
-0.0274	PoCv	0.0997	PoCf
0.0111	PoFI	0.2412	Mem

error in predicting the correct rank is less important than the accuracy of the models in correctly identifying the top ten percent, twenty percent and so on.

The average percent correct for the predictions made by the models using the training data from Mp3gain and the test data from Mv is shown in Table 4.21. As expected, the overall accuracy is lower than with the cross-validation, because the training set used is based on a single software project, so the relationships between the attributes and the class are particular to that instance. Note that zeroR and Kstar are all zero in this case because they determined a single value and predicted that single value for each instance, leaving no correctly identified instances. Alternatively, the least median squared algorithm gets much

Table 4.18: Validation Relief Attribute Evaluation

Influence	Attribute	Influence	Attribute
0.747	PoFI	-0.0195	Mental
0.7413	VarFreq	-0.0204	Vocab
0.7413	PoEv	-0.0209	n1
0.6993	Reads	-0.0213	n2
0.2188	Writes	-0.0216	Complexity
0.2151	vartype	-0.0224	LogLoc
0.2005	LC	-0.0226	ModLoC
0.1755	size	-0.0229	Mem
0.1755	PoCv	-0.0229	PoCf
-0.0138	PoEf	-0.0274	AvgNested
-0.0138	ELoC	-0.0291	MaxNested
-0.0149	Difficulty	-0.0308	Propagation
-0.0189	N1	-0.0377	FuncFreq
-0.0191	Length	-0.0405	Branches
-0.0193	Effort	-0.045	IMC
-0.0193	Volume	-0.055	Comments
-0.0194	N2		

better results in the twenty percent range and beyond. The IBk and the M5P algorithms fall behind the least median squared algorithm in their predictive ability, catching up only when determining the bottom percent ranges of eighty percent of variables or higher. Linear regression is particularly bad at dealing with this test case. It is interesting that linear regression does poorly in this case, but the least median squared algorithm which uses linear regression for predicting numeric classes does quite well compared to the other baseline models.

When the datasets are combined into the single dataset for validating through cross-validation, the models saw an increase in predicting ability because of the training process incorporating data points from numerous software projects which provides a better informed model. IBk returns to being the best model in this case, but least median squares is not far behind in accuracy. The most interested data is the in top ten to thirty percent prediction ranges for variables, because the user is most likely interested in upping the error handling on

Table 4.19: Mv Variable Evaluation

Model	Total Mean Error	Mean Absolute Error	Root Mean Squared Error
zeroR	0.94	37.95	385.71
Linear Regression	1.85	33.58	333.58
Ibk	1.35	33.16	334.08
Kstar	0.94	37.95	385.71
LWL	1.95	26.09	275.95
LeastMedSq	0.41	10.74	123.27
M5P	0.97	27.16	273.67

Table 4.20: Validation Variable Evaluation

Model	Total Mean Error	Mean Absolute Error	Root Mean Squared Error
zeroR	3.07	28.26	944.34
Linear Regression	1.03	736.10	736.10
Ibk	0.35	6.20	243.79
Kstar	2.16	17.14	614.65
LWL	1.49	22.40	779.65
LeastMedSq	0.48	7.89	317.01
M5P	0.77	22.13	725.14

those areas. Overall it appears that least median squared is the best choice for predictions for relative importance of variables, despite performing worse than IBk in the validation dataset. The cross-validation is a good way to measure performance of a smaller sample size that we have in this research, but it is important to measure the capability of the models when dealing with potentially unknown data trends from software projects with different architectures. Figure 4.4 shows a representation of the prediction accuracy of the different models tested against the Mv dataset usign the Mp3gain dataset for training. Figure 4.5 shows the same, but for the combined validation dataset using cross-validation.

The following data is representative of the average difference in the actual ranks of the variables in the datasets and the ranks that the predictive model chose for each of the variables. Table 4.23 shows the average difference across each model for the different

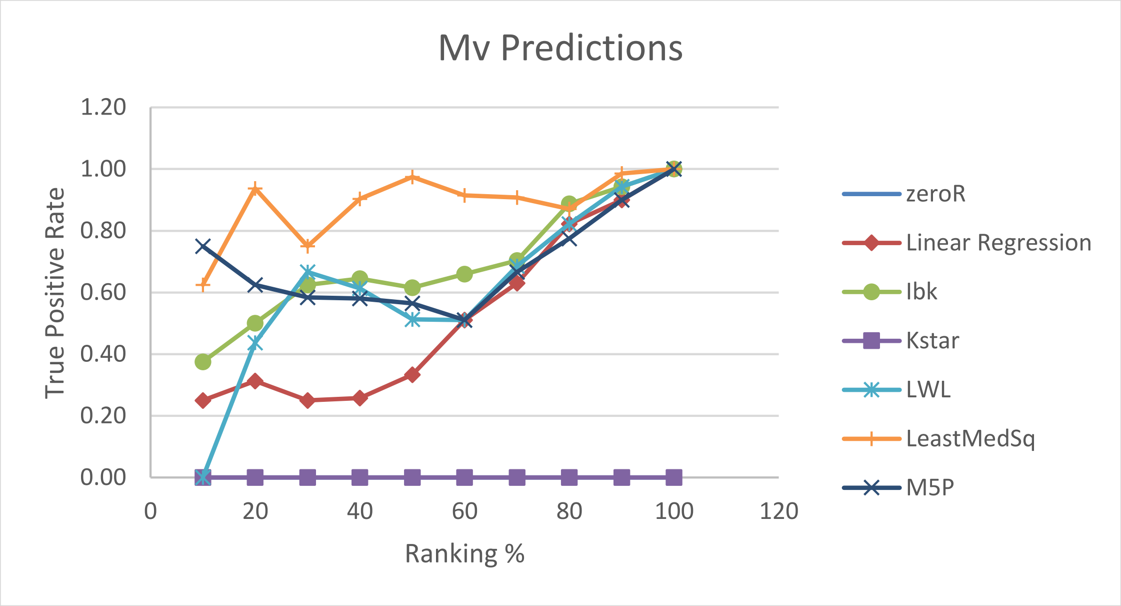

Figure 4.4: Mv Predictions

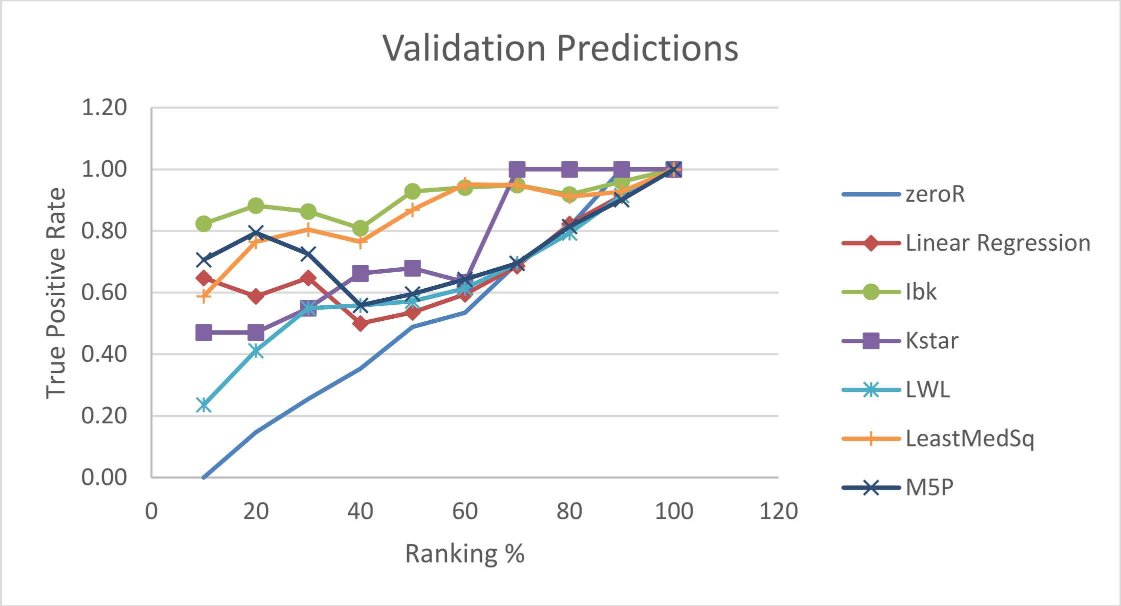

Figure 4.5: Validation Predictions

Table 4.21: Mv Variable Average Percent Correct

Percentage	zeroR	Linear Regression	Ibk	Kstar	LWL	LeastMedSq	M5P
10	0.00	0.25	0.38	0.00	0.00	0.63	0.75
20	0.00	0.31	0.50	0.00	0.44	0.94	0.63
30	0.00	0.25	0.63	0.00	0.67	0.75	0.58
40	0.00	0.26	0.65	0.00	0.61	0.90	0.58
50	0.00	0.33	0.62	0.00	0.51	0.97	0.56
60	0.00	0.51	0.66	0.00	0.51	0.91	0.51
70	0.00	0.63	0.70	0.00	0.69	0.91	0.67
80	0.00	0.82	0.89	0.00	0.82	0.87	0.77
90	0.00	0.90	0.94	0.00	0.94	0.99	0.90
100	0.00	1.00	1.00	0.00	1.00	1.00	1.00

Table 4.22: Validation Variable Average Percent Correct

Percentage	zeroR	Linear Regression	Ibk	Kstar	LWL	LeastMedSq	M5P
10	0.00	0.65	0.82	0.47	0.24	0.59	0.71
20	0.15	0.59	0.88	0.47	0.41	0.76	0.79
30	0.25	0.65	0.86	0.55	0.55	0.80	0.73
40	0.35	0.50	0.81	0.66	0.56	0.76	0.56
50	0.49	0.54	0.93	0.68	0.57	0.87	0.60
60	0.53	0.59	0.94	0.63	0.61	0.95	0.64
70	0.69	0.69	0.95	1.00	0.69	0.95	0.69
80	0.81	0.82	0.92	1.00	0.79	0.91	0.81
90	1.00	0.91	0.96	1.00	0.91	0.93	0.90
100	1.00	1.00	1.00	1.00	1.00	1.00	1.00

percentage ranges. The data here is representative of the same trends found in the percent correct table detailed above. The models that have lower correct predictions also have a greater variance in the predictions made by that model. For example, the least median squared algorithm had the best prediction rates and also the lowest variance in the actual and predicted ranks for each of the variables with respect to the relative importance of each variable. The models that provided a lower true positive prediction rate also had a higher difference on average between the actual ranks and the predicted ranks of the variables. Again, zeroR and Kstar have the same values because they both pick threshold values and

Table 4.23: Mv Variable Average Rank Difference

Percentage	zeroR	Linear Regression	Ibk	Kstar	LWL	LeastMedsq	M5P
10	3.50	33.75	23.38	3.50	27.38	6.75	7.63
20	7.50	37.88	29.44	7.50	23.88	6.25	19.31
30	11.50	35.46	24.92	11.50	23.21	6.21	20.88
40	15.00	31.68	24.71	15.00	24.81	6.35	21.94
50	19.00	30.23	25.18	19.00	25.46	6.26	22.62
60	23.00	29.09	26.26	23.00	23.43	7.43	23.00
70	27.46	30.06	28.52	27.46	23.87	8.13	23.69
80	31.45	30.61	30.95	31.45	24.47	8.69	24.06
90	34.44	32.30	32.90	34.44	26.26	10.09	25.60
100	36.96	33.14	33.12	36.96	26.06	10.73	26.81

attempt to match the variables to those as opposed to calculating a unique prediction for each variable. In the top thirty percent of the variable rankings, least median squared, M5P, and Ibk had the best overall results, because the Kstar and zeroR model results are not usable for actual ranking.

The validation dataset results are shown in Table 4.24 and reinforce the trend shown in the previous data for the validation dataset. For the combined dataset, IBk gives the best results, then least median squares and M5P in a distant third. Similarly to the prediction metrics, the overall best model in this experiment is least median squared. When comparing the results of the model predictions against an unseen dataset and the combined dataset, least median squares performs better overall, with IBk outperforming on the combined set, but falling far behind for the dataset from the Mv program. Figure 4.6 shows the curve of the average difference in actual and predicted rank for the models. Figure 4.7 shows the same relationship but on the data from the validation dataset.

4.4 Summary

The results of the baseline testing on Mp3gain show that machine learning is an effective choice for increasing the efficiency of the framework by correctly predicting the importance of

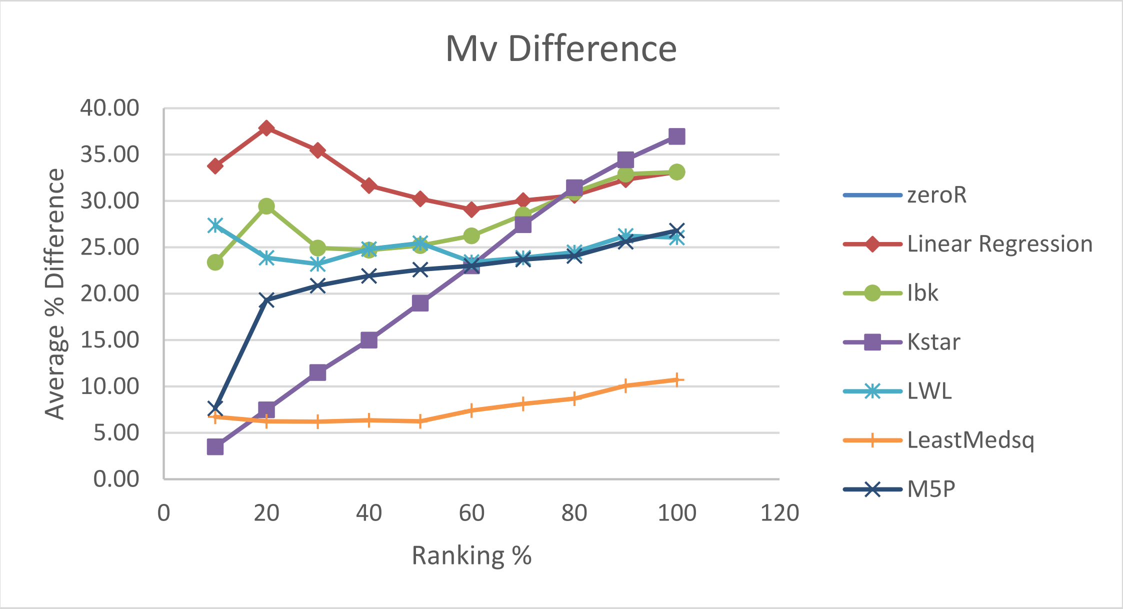

Figure 4.6: Mv Average Percent Difference

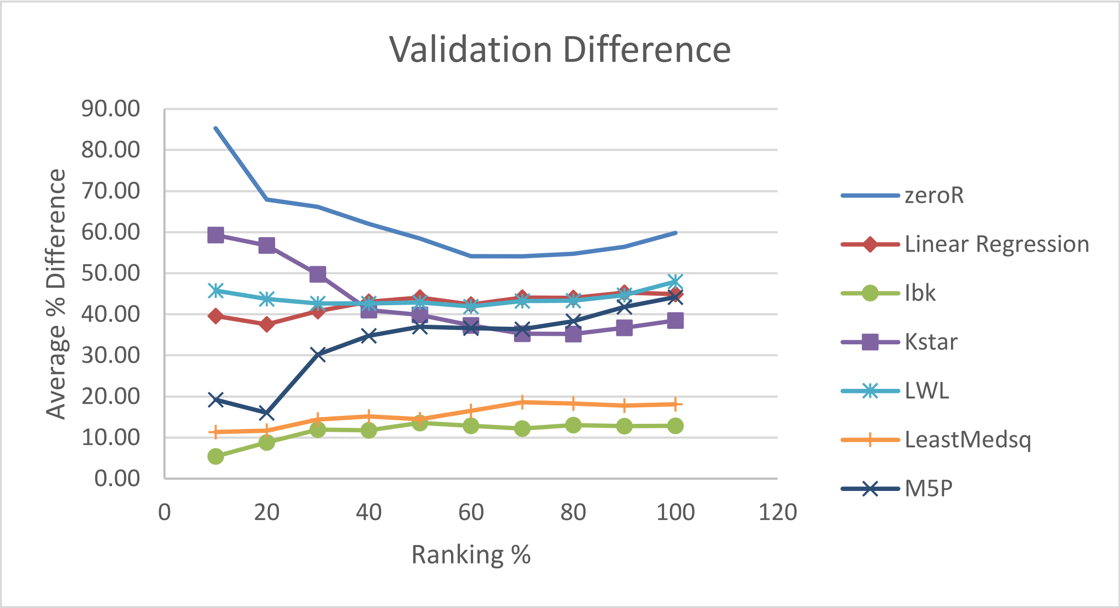

Figure 4.7: Validation Average Percent Difference

Table 4.24: Validation Variable Average Rank Difference

Percentage	zeroR	Linear Regression	Ibk	Kstar	LWL	LeastMedsq	M5P
10	85.29	39.59	5.41	59.29	45.76	11.35	19.24
20	67.91	37.56	8.79	56.76	43.71	11.71	16.03
30	66.18	40.82	11.90	49.75	42.69	14.35	30.24
40	62.03	43.06	11.76	41.04	42.60	15.15	34.74
50	58.48	44.06	13.57	39.89	42.86	14.49	36.99
60	54.13	42.43	12.89	37.35	41.91	16.50	36.62
70	54.12	44.06	12.21	35.31	43.27	18.60	36.37
80	54.73	44.02	13.06	35.22	43.32	18.28	38.36
90	56.39	45.28	12.82	36.76	44.67	17.74	41.78
100	59.86	44.90	12.86	38.51	47.96	18.11	44.20

functions and variables. A combination of standard software metrics and the relative metrics introduced in this work need to be used in order to make the best possible prediction. In addition to these metrics, some general aspects of the code that are used as features in the dataset are particularly important in the prediction process. For example, the type of variable has a major impact on the importance of the variable for data faults in the system. Pointer variables, including strings, are in general more indicative of higher risk variables in a system due to their more complex nature in how they are stored and utilized in memory. The number of writes occurring for a particular variable was another indicator of high risk variables. A greater focus on the real probabilities of the occurrence of a data fault and its execution was an effective way to measure the importance of variables in the system. The high impact of the propagation metric for predicting function priority indicates that the architecture of the system and data-flow has a high impact on the priority of a function in a system.

Of the various machine learning pre-processing methods, cost benefit analysis proved to be the most effective at increasing the true positive prediction rate. Though the overall percentage of correct predictions lowered in some cases, the cost-benefit analysis increased the total number of true positives. For efficient reliability verification we can afford to have some false positives, but false negatives could lead to letting a potentially high risk function go by untested. Discretization of the attributes in the datasets did not provide a consistent improvement to the prediction capabilities of the models. Using the Multilayer Perceptron and J48 models for function prediction saw true positive prediction rates of over eight-percent, as high as eighty-seven percent with Multilayer Perceptron. It should be noted that overall between both cross-validation and test data, J48, a tree algorithm, performed the best for the function model.

The best algorithms for variable prediction were Least Median Squared, IBk and M5P. In the case of variable prediction, Least Median Squared was far ahead of the others, with eighty to ninety percent correctly ranked variables in the top ten to thirty percent of rankings. Even when dealing with the fresh datasets and the accuracy drop, Least Median Squared showed an average accuracy of seventy-seven percent against the top thirty percent of unseen datapoints. Comparatively, the lazy and rule based algorithms had drastic drops when dealing with completely unseen datapoints. Given that it is unlikely that it will be possible to collect enough training data for these models to have a rule for every scenario, it is best to pick a model that gracefully handles unseen data points. Least Median Squared is the best overall model for dealing with ranking the variables according to relative importance. This is the case despite the cross-validation showing better results with IBk, because the lazy model is on average twenty percent worse than Least Median Squared on unseen datapoints.

Chapter 5

Conclusion

This dissertation is a valuable step towards utilizing machine learning algorithms in combination with software metrics to highlight the highest priority functions and variables in an open-source software system. Both historically accepted software metrics and the newer relative software metrics have proven to be effective features for the machine learning algorithms. The experimental results indicate a high potential for an effective means of establishing the highest priority variables in open-source software by utilizing a combination of static and dynamic analysis with machine learning methods. The results serve as a good indicator of the trends in software metrics when applied to open-source software tested with relative use cases. A much larger selection of machine learning algorithms was reduced in size to the smaller subset of machine learning algorithms presented. The results indicate the prevalence of function and tree based machine learning algorithms for predicting the relative importance of both functions and variables. New metrics need to be introduced to replace those that were consistently eliminated by the attribute selection process or are redundant with other metrics. The next metrics to be tested will be those that can be directly connected to the existing testing process for identifying key functions in the system. Afferent and efferent coupling and cohesion measurements make sense, given the high ranking of the propagation metric.

It is important to note that this work was based on two software projects with different architectures and coding strategies. By running more software projects through the framework and building larger datasets, the predictions should get increasingly more accurate. This is supported by the learning curve of the model predictions and the cross-validation results with the combined dataset. With the architecture agnostic approach taken in this research, the focus was on the relationship of the variables and functions to the rest of the target software. The current dataset built around only a couple of programs will fall short on accurately predicting functions and variables in new datasets. The cross-validation results show that the concept of utilizing machine learning to predict the relative importance of the variables in a system is a valid approach and worth pursuing further. The mv test cases that utilize the Mp3gain datasets for training had the expected results, but make the need for varying types data more prominent. With the addition of new data, these drawbacks of using the models on new datasets should lower significantly, as shown by the cross-validation. This research focuses on a more general view of open-source software systems. This research has proven to be a potentially powerful tool to use when dealing with integrating open-source or any post-release stage code where the source code is available.

The addition of machine learning and the updates to the framework have proven to be an effective way to increase the efficiency of the reliability verification process. The research shows a significant reduction in the total time needed to prioritize the high risk locations in the code. The two stage predictive process presents two models which are capable of further reducing the time needed to verify the reliability of a software system compared to [1]. We see a further reduction in the total number of test cases when you consider that each of the variables does not need fault injection testing to be done for the ranking. The next logical step is to pursue even greater accuracy from the predictive models to make the framework a more desirable process for completing reliability verification in high fidelity systems. The learning curve and data from the cross-validation suggests that further improvements on the effectiveness can be made with additional research into training and testing the machine learning models. With the improvements discussed in the future work section, a completed tool should increase the overall efficiency of verifying the reliability of software under the effects of data faults.

Chapter 6

Future Work

The individual components of the framework utilized in this dissertation research have been tested and validated using two sample software systems. There is further research to be done in order to get the framework to a point where it can be utilized in a real-world situation. The overall process needs to be fully automated from start to finish to allow for a more efficient means of extracting features from the target software for analysis. Though not specifically a research problem to be solved, there is a technical aspect of automating the framework that is non-trivial. Once this automation of the framework is complete, the efficiency of the testing and analysis process will increase significantly, allowing for me to build datasets for training the models much faster.

Automation of the static and dynamic analysis and the machine learning portions of the process will lead to an overall more efficient process for prioritizing test cases. This automation is the next milestone for this research. Until the framework is fully automated, gathering new data, which has proven to be a vital aspect of improving both the classification and regression models, will be a cumbersome and inefficient process overall. The final stage of the framework would ideally by a tool that could be used on a piece of software to complete each of the stages of the process from defining the critical path of priority modules, functions and variables up to the placement of the error handling mechanisms. There is a great deal of parsing, logging and interaction between the various components of the framework that need to be coded and tested. The target software would need to be run through the static and dynamic analysis sections which complete their calculation of the remaining metrics and that data should be used to create a dataset and automatically fed into a machine learning algorithm to be tested against a trained model.

A big technical challenge would be automatically instrumenting the code in order to complete the fault injection testing and to obtain some of the dynamic analysis metrics, such as variable size. This obtrusive way of conducting the fault injection analysis will be changed in the future, after achieving automation of the testing and analysis framework. The framework needs to move away from the limitations of testing and being run using only C source code. The framework will likely move to a more portable language, such as python, in order to incorporate both the scripting elements of the framework and the analysis and testing aspects. With libraries like Cython, the original C source code for the fault injection component can be maintained while being integrated into the new python coded framework. In addition to changing the framework source code to a python base, the framework needs to be revised and improved upon to allow for more than just testing and analysis on C source code. This will involve adding new code to be utilized based on the language the target source code is written in, but also a change in the functionality of some of the components of the framework.

Of particular concern is the implementation of the fault injection component of the framework. The component works fine for this first build of the framework for the greater problem of creating a more efficient process of identifying high risk variables in a system for reliability verification. But, the implementation of the fault injection framework will either need to be expanded to have probe and injection types for each type of source language that will be testing using the framework, or the implementation needs to change. Ideally, the fault injection component would avoid the intrusive style of injection that involves directly injecting through calls in the source code, to utilizing system calls in a simulation type environment that monitors the source code while it executes to detect the creating of variables and directly injects the data fault into the memory location that the variable value was saved into. Completing the injection in this way gives the framework more credibility for use on real-world systems and allows for a more flexible design for dealing with varying types of source languages for the target software.

Another task is taking the fully automated framework and making it into one that is much easier to use than the disjointed command line tools and scripts that currently make up the process. The addition of components such as a GUI would allow for the user to easily point the framework to the target software system and have it complete the testing and analysis based on some inputs the user gives as to any considerations the user may have for how to complete the tests. An example would be the user providing the relative use cases for a command-line driven system, meaning the list of commands that the user is interested in testing the software under. Interoperability with well known output types, such as CSV or XML, for datasets and ranking outputs is the next step in this process of usability. The user will want to store the output of the analysis and testing in a common output to allow for ease of use. It is clear that there are a number of improvements that can be made to the framework created during this dissertation research. The main components have been researched and validated and now must be made into a more user friendly and efficient process. The improvements detailed in this section are aimed at improving the overall experience in utilizing the framework, but also increasing its efficiency in completed testing in order to more rapidly build the datasets that will be used to train the machine learning models.

The research has brought up some important questions that need to be pursued further in order to get the framework to the point where it can be used on a real-world system. The machine learning algorithms highlighted some key trends in the software dealing with which features were most important for making a correct prediction. Another aspect to these trends was the difference in the trends found for each of the target software systems. It would be worthwhile to research the trends in the metrics for predicting relative importance based on the architecture of the target system. The architecture of the system appears to have an effect on relationships of the metrics to each other.

After building up the training dataset, it would be interesting to split the current data into smaller datasets based on the architecture of the software system it came from. The prediction capabilities of the models could be compared across all of the datasets for a software system. It would make sense for the predictions to be better when the model was trained on data that came from software systems with a similar architecture to the current target software being tested. If this is the case, it would be beneficial to have a collection of training data that can be used to making the predictions based on the data-flow analysis of the target software.

Another aspect of the research that could be further explored is the approach to simulating the data faults occurring in the target system. A comparison of the different ways to cause the data fault in the system would give valuable data for updating the framework for use in real-world systems. The simulation should mimic as closely as possible how the data faults occur in reality. Utilizing a method to directly access memory to change the bits instead of doing it through the source code could lead to a more effective fault injection process. The goal is to have as flexible a framework as possible to allow for more widespread use. If the framework was based around system calls and direct memory access, there would be fewer versions of the fault injection. With injections based around the language used to program the source code, there are a significant number of languages that would need to be made compatible.

With the goal of creating a framework that simulates the faults as close to reality as possible, it may be beneficial to pursue a hardware component to the testing. Using embedded programs as an example, it may be possible to run software and physically cause data faults in the system. The locations of the faults could be recorded and compared to the effect that it had on the system. Pursuing this line of research may be useful in validating the methods used in this work for calculating the probability of corruption and execution. Also, by utilizing some physical component to the testing, it may assuage the concerns of those less familiar with the software implementation and give a more concrete test case for use in a real-world scenario.

Bibliography

[1] Pape, P. 2013. *A Methodology for Increasing the Dependability of Open Source Software Components*. Master's thesis. Auburn University, Auburn, AL.

[2] Ayala, C. P., Cruzes, D. S., Hauge, O., Conradi, R. (2011). Five facts on the adoption of open source software. *Software, IEEE*, 28(2), 95-99.

[3] Fitzgerald, B., Kesan, J. P., Russo, B., Shaikh, M., Succi, G. (2012). Lessons learned from the adoption of open source software. *European Financial Review*. url = http://hdl.handle.net/10344/2423

[4] Gwebu, K. L., Wang, J. (2011). Adoption of Open Source Software: The role of social identification. *Decision Support Systems*, 51(1), 220-229.

[5] Reed, A. (2011). Negative Scarcity And The Praxeology Of Open Source Software. *Journal of Business and Economics Research (JBER)*, 6(2).

[6] Spinellis, D., Giannikas, V. (2012). Organizational adoption of open source software. *Journal of Systems and Software*, 85(3), 666-682.

[7] Hecker, F. (1999). Setting up shop: the business of Open-Source software. *IEEE software*, 16(1), 45-51.

[8] Kumar, V., Gordon, B. R., and Srinivasan, K. (2011). Competitive strategy for open source software. *Marketing Science*, 30(6), 1066-1078.

[9] Bonaccorsi, A., Rossi, C. (2003). Why open source software can succeed. *Research policy*, 32(7), 1243-1258.

[10] Colombo, M. G., Piva, E., Rossi-Lamastra, C. (2013). Open innovation and within-industry diversification in small and medium enterprises: The case of open source software firms. *Research Policy*.

[11] Li, P. L., Kivett, R., Zhan, Z., Jeon, S. E., Nagappan, N., Murphy, B., Ko, A. J. (2011, May). Characterizing the differences between pre-and post-release versions of software. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 716-725). ACM.

[12] Midha, V., Palvia, P. (2012). Factors affecting the success of Open Source Software. *Journal of Systems and Software*, 85(4), 895-905.

[13] West, J., Gallagher, S. (2006). Challenges of open innovation: the paradox of firm investment in open?source software. *R and D Management*, 36(3), 319-331.

[14] Cotroneo, D., Grottke, M., Natella, R., Pietrantuono, R., Trivedi, K. S. (2013, November). Fault triggers in open-source software: An experience report. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on* (pp. 178-187). IEEE.

[15] Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C. (2006). Have things changed now?. In *An empirical study of bug characteristics in modern open source software. In Proceedings of the ASID Workshop in ASPLOS.*

[16] bin Saleem, S., Yu, Y., Nuseibeh, B., Ince, D., Lopez, T. (2012). An Empirical Study of Security Requirements in Planning Bug Fixes for an Open Source Software Project. url = http://computing-reports.open.ac.uk/2012/TR2012-01.pdf

[17] Yin, Z., Caesar, M., Zhou, Y. (2010). Towards understanding bugs in open source router software. *ACM SIGCOMM Computer Communication Review*, 40(3), 34-40.

[18] Aberdour, M. (2007). Achieving quality in open-source software. *Software, IEEE*, 24(1), 58-64.

[19] Crowston, K., Wei, K., Howison, J., Wiggins, A. (2012). Free/Libre open-source software development: What we know and what we do not know. *ACM Computing Surveys (CSUR)*, 44(2), 7.

[20] Gyimothy, T., Ferenc, R., Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10), 897-910.

[21] Yuan, G., Gygi, F. (2012). A distributed approach to verification and validation of electronic structure simulation data using ESTEST. *Computer Physics Communications*, 183(8), 1744-1748.

[22] Chaturvedi, K. K., Bedi, P., Misra, S., Singh, V. B. (2013, December). An Empirical Validation of the Complexity of Code Changes and Bugs in Predicting the Release Time of Open Source Software. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on* (pp. 1201-1206). IEEE.

[23] Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D. (2013). Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1), 53-95.

[24] Shin, Y., Meneely, A., Williams, L., Osborne, J. A. (2011). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *Software Engineering, IEEE Transactions on*, 37(6), 772-787.

[25] Sinha, B. R., Dey, P. P., Amin, M., Badkoobehi, H. (2013). Software complexity measurement using multiple criteria. *Journal of Computing Sciences in Colleges*, 28(4), 155-162.

[26] Thomas, S. W., Hemmati, H., Hassan, A. E., Blostein, D. (2014). Static test case prioritization using topic models. *Empirical Software Engineering*, 19(1), 182-212.

[27] Zhang, H., Zhang, X., Gu, M. (2007, December). Predicting defective software components from code complexity measures. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on* (pp. 93-96). IEEE.

[28] Cadar, C., Dunbar, D., Engler, D. (2008). KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. Stanford University

[29] Cadar, C., Godefroid, P., Khurshid, S., P?s?reanu, C. S., Sen, K., Tillmann, N., Visser, W. (2011, May). Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 1066-1071). ACM.

[30] Cadar, C., Sen, K. (2013). Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2), 82-90.

[31] Kaushik, N., Salehie, M., Tahvildari, L., Li, S., Moore, M. (2011, March). Dynamic prioritization in regression testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on* (pp. 135-138). IEEE.

[32] Mishra, K. K., Kumar, A., Misra, A. K. (2014). A novel approach for minimizing and prioritizing test suite. *Engineering Science Letters*, 2014, Article-ID.

[33] Li, Y., Su, Z., Wang, L., Li, X. (2013, October). Steering symbolic execution to less traveled paths. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages and applications* (pp. 19-32). ACM.

[34] Marinescu, P. D., Candea, G. (2011). Efficient testing of recovery code using fault injection. *ACM Transactions on Computer Systems (TOCS)*, 29(4), 11.

[35] Marinescu, P. D., Cadar, C. (2012). High-coverage symbolic patch testing. In *Model Checking Software* (pp. 7-21). Springer Berlin Heidelberg.

[36] Marinescu, P. D., Cadar, C. (2013, August). KATCH: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (pp. 235-245). ACM.

[37] Mirzaaghaei, M., Pastore, F., Pezz, M. (2014). Automatic test case evolution. *Software Testing, Verification and Reliability.*

[38] Schrammel, P., Melham, T., Kroening, D. (2013). Chaining Test Cases for Reactive System Testing. In *Testing Software and Systems* (pp. 133-148). Springer Berlin Heidelberg.

[39] Siddiqui, J. H., Khurshid, S. (2012, October). Scaling symbolic execution using ranged analysis. In *ACM SIGPLAN Notices* (Vol. 47, No. 10, pp. 523-536). ACM.

[40] Siddiqui, J. H., Khurshid, S. (2012, March). Staged symbolic execution. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (pp. 1339-1346). ACM.

[41] Siddiqui, J. H., Khurshid, S. (2013). Scaling symbolic execution using staged analysis. *Innovations in Systems and Software Engineering*, 9(2), 119-131.

[42] Yoo, S., Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2), 67-120.

[43] Benameur, A., Evans, N. S., Elder, M. C. (2013). MINESTRONE: Testing the SOUP. In *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test.* USENIX.

[44] Bryce, R. C., Sampath, S., Memon, A. M. (2011). Developing a single model and test prioritization strategies for event-driven software. *Software Engineering, IEEE Transactions on*, 37(1), 48-64.

[45] Carlson, R., Do, H., Denton, A. (2011, September). A clustering approach to improving test case prioritization: An industrial case study. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on* (pp. 382-391). IEEE.

[46] Czerwonka, J., Das, R., Nagappan, N., Tarvo, A., Teterev, A. (2011, March). Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on* (pp. 357-366). IEEE.

[47] Jacob, T. P., Ravi, T. (2013). Detecting of software source code defects using test case prioritization rules. In *2nd International Conference on Latest Computational Technologies.*

[48] Jiang, B., Chan, W. K., Tse, T. H. (2011, July). On practical adequate test suites for integrated test case prioritization and fault localization. In *Quality Software (QSIC), 2011 11th International Conference on* (pp. 21-30). IEEE.

[49] Jiang, B., Zhang, Z., Chan, W. K., Tse, T. H., Chen, T. Y. (2012). How well does test case prioritization integrate with statistical fault localization?. *Information and Software Technology*, 54(7), 739-758.

[50] Wei, J., Rashid, L., Pattabiraman, K., Gopalakrishnan, S. (2011, June). Comparing the effects of intermittent and transient hardware faults on programs. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on* (pp. 53-58). IEEE.

[51] Zhou, Y., and Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *Software Engineering, IEEE Transactions on, 32*(10), 771-789.

[52] Suresh, Y., Kumar, L., and Rath, S. K. (2014). Statistical and Machine Learning Methods for Software Fault Prediction Using CK Metric Suite: A Comparative Analysis. *International Scholarly Research Notices*, 2014.

[53] Malhotra, R., and Jain, A. (2012). Fault Prediction Using Statistical and Machine Learning Methods for Improving Software Quality. *JIPS, 8*(2), 241-262.

[54] Jiang, Y., Cuki, B., Menzies, T., and Bartlow, N. (2008, May). Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th international workshop on Predictor models in software engineering* (pp. 11-18). ACM.

[55] Shepperd, M., Bowes, D., and Hall, T. (2014). Researcher bias: The use of machine learning in software defect prediction. *Software Engineering, IEEE Transactions on, 40*(6), 603-616.

[56] Gondra, I. (2008). Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software, 81*(2), 186-195.

[57] Catal, C. (2011). Software fault prediction: A literature review and current trends. *Expert systems with applications, 38*(4), 4626-4636.

[58] Gao, K., Khoshgoftaar, T. M., Wang, H., and Seliya, N. (2011). Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience, 41*(5), 579-606.

[59] Wang, H., Khoshgoftaar, T. M., and Seliya, N. (2011, May). How many software metrics should be selected for defect prediction?. In *FLAIRS Conference.*

[60] Kitchenham, B. (2010). What's up with software metrics? preliminary mapping study. *Journal of systems and software, 83*(1), 37-51.

[61] Gray, D., Bowes, D., Davey, N., Sun, Y., and Christianson, B. (2011, April). The misuse of the nasa metrics data program data sets for automated software defect prediction. In *Evaluation and Assessment in Software Engineering (EASE 2011), 15th Annual Conference on* (pp. 96-103). IET.

[62] Nori, A. V. (2014, June). Software reliability via machine learning (invited talk). *In Proceedings of the 2nd FME Workshop on Formal Methods in Software Engineering* (pp. 1-2). ACM.

[63] Boehm, B. W. (1981). *Software engineering economics* (Vol. 197). Englewood Cliffs (NJ): Prentice-hall.

[64] Imsand, E. S., Evans, G., Dozier, G., Hamilton, J. A. (2004). Using genetic algorithms to aid in a vulnerability analysis of national missile defense simulation software. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology, 1*(4), 215-223.

[65] Kapur, P. K., Pham, H., Anand, S., Yadav, K. (2011). A unified approach for developing software reliability growth models in the presence of imperfect debugging and error generation. *Reliability, IEEE Transactions on, 60*(1), 331-340.

[66] Holmberg, J. E., Bishop, P., Guerra, S., Thuy, N. (2012, June). Safety case framework to provide justifiable reliability numbers for software systems. *In Proceedings of 11th International Probabilistic Safety Assessment and Management Conference and The Annual European Safety and Reliability Conference.*

[67] Qiuying, L., Lei, L. (2013). Determining the Minimal Software Reliability Test Effort by Stratified Sampling. *TELKOMNIKA Indonesian Journal of Electrical Engineering, 11*(8), 4399-4406.

[68] Veevers, A. (2014, June). Software Coverage Metrics and Operational Reliability. *In Safety of Computer Control Systems 1990 (SAFECOMP'90): Proceedings of the IFAC/EWICS/SARS Symposium* Gatwick, UK, 30 October-2 November 1990 (p. 67). Elsevier.

[69] Vouk, M., Williams, L. (2013, November). Using software reliability models for security assessment verification of assumptions. *In Software Reliability Engineering Workshops* (ISSREW), *2013 IEEE International Symposium on* (pp. 23-24). IEEE.

[70] Jin, C., Jin, S. W. (2014). Software reliability prediction model based on support vector regression with improved estimation of distribution algorithms. *Applied Soft Computing*, 15, 113-120.

[71] Wiper, M. P., Palacios, A. P., Marin, J. (2012). Bayesian software reliability prediction using software metrics information. *Quality Technology and Quantitative Management, 9*(1), 35-44.

[72] Pati, J., Shukla, K. K. (2015, February). A Hybrid Technique for Software Reliability Prediction. *In Proceedings of the 8th India Software Engineering Conference* (pp. 139-146). ACM.

[73] Hu, H., Jiang, C. H., Cai, K. Y., Wong, W. E., Mathur, A. P. (2013). Enhancing software reliability estimates using modified adaptive testing. *Information and Software Technology, 55*(2), 288-300.

[74] Hsu, C. J., Huang, C. Y. (2011). An adaptive reliability analysis using path testing for complex component-based software systems. *Reliability, IEEE Transactions on, 60*(1), 158-170.

[75] Bhuyan, M. K., Mohapatra, D. P., Sethi, S. (2014). A survey of computational intelligence approaches for software reliability prediction. *ACM SIGSOFT Software Engineering Notes, 39*(2), 1-10.

[76] Ma, C., Gu, G., Zhao, J. (2012). A Novel Software Reliability Assessment Approach based on Neural Network in Network Environment. *IJACT: International Journal of Advancements in Computing Technology, 4*(1), 136-144.

[77] `http://linux.die.net/man/1/gcov` - gcov Linux manual page

[78] `http://linux.about.com/library/cmd/blcmdl1_nm.htm` - nm Linux/Unix Command man page

[79] `http://linux.about.com/library/cmd/blcmdl1_objdump.htm` - objdump Linux/Unix Command man page

[80] `http://www.stack.nl/~dimitri/doxygen/` - Doxygen main page

[81] `http://graphviz.org` - graphviz main page

[82] Chhabra, P., and Bansal, L. (2014). An Effective Implementation of Improved Halstead Metrics for Software Parameters Analysis.

[83] McCabe, T. J. (1976). A complexity measure. *Software Engineering, IEEE Transactions on*, (4), 308-320.

[84] `http://www.gnu.org/software/gdb/` - gdb: The GNU Project Debugger

[85] Leeke, M., Jhumka, A. Towards Understanding the Importance of Variables in Dependable Software. *in Dependable Computing Conference* (EDCC). (Valencia, Spain 2010) 85-94.

[86] Holmes, G., Donkin, A., Witten, I. H. (1994, December). Weka: A machine learning workbench. *In Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on* (pp. 357-361). IEEE.

[87] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I. H. (2009). The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter, 11*(1), 10-18.

[88] Kononenko, I. (1995, August). On biases in estimating multi-valued attributes. In *IJCAI* (Vol. 95, pp. 1034-1040).

[89] Fayyad, U., and Irani, K. (1993). Multi-interval discretization of continuous-valued attributes for classification learning.

[90] Quinlan, J. R. (1993). *C4.5: programs for machine learning* (Vol. 1). Morgan Kaufmann Publishers.

[91] John, G. H., and Langley, P. (1995, August). Estimating continuous distributions in Bayesian classifiers. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence* (pp. 338-345). Morgan Kaufmann Publishers Inc.

[92] Aha, D. W., Kibler, D., and Albert, M. K. (1991). Instance-based learning algorithms. *Machine learning, 6*(1), 37-66.

[93] Kenji Kira, Larry A. Rendell: A Practical Approach to Feature Selection. *In: Ninth International Workshop on Machine Learning*, 249-256, 1992.

[94] John G. Cleary, Leonard E. Trigg: K*: An Instance-based Learner Using an Entropic Distance Measure. *In: 12th International Conference on Machine Learning*, 108-114, 1995.

[95] Eibe Frank, Mark Hall, Bernhard Pfahringer: Locally Weighted Naive Bayes. *In: 19th Conference in Uncertainty in Artificial Intelligence*, 249-256, 2003.

[96] Peter J. Rousseeuw, Annick M. Leroy (1987). Robust regression and outlier detection.

[97] Peter J. Rousseeuw, Least Median of Squares regression. *Journal of the American Statisctical Association*, December 1984, 79(388)

[98] Ross J. Quinlan: Learning with Continuous Classes. *In: 5th Australian Joint Conference on Artificial Intelligence*, Singapore, 343-348, 1992.

[99] Y. Wang, I. H. Witten: Induction of model trees for predicting continuous classes. *In: Poster papers of the 9th European Conference on Machine Learning*, 1997.

[100] M. A. Hall (1998). Correlation-based Feature Subset Selection for Machine Learning. Hamilton, New Zealand.

Appendices

Appendix A

Code Listing

```c
//Patrick Pape - 4/13 - Header file for injector functions

#ifndef INJECTOR_H_INCLUDED
#define INJECTOR_H_INCLUDED

#include <stdbool.h>

//Create the list of injectors for the experiment - file is taken from simulator input
void injector_setup();

//Injector function to inject a fault if it is part of the experiment list
char injector_c(char *iid, char var, char inject_val, float p);
char *injector_s(char *iid, char *var, char *inject_val, float p);
int injector_i(char *iid, int var, int inject_val, float p);
short injector_sh(char *iid, short var, short inject_val, float p);
long injector_l(char *iid, long var, long inject_val, float p);
float injector_f(char *iid, float var, float inject_val, float p);
double injector_d(char *iid, double var, double inject_val, float p);
long double injector_ld(char *iid, long double var, long double inject_val, float p);
//bool injector_b(char *iid, bool var, bool inject_val, float p);

//Injector function to inject a random fault if it is part of the experiment list
char injector_rand_c(char *iid, char var, float p);
char *injector_rand_s(char *iid, char *var, float p);
int injector_rand_i(char *iid, int var, float p);
short injector_rand_sh(char *iid, short var, float p);
long injector_rand_l(char *iid, long var, float p);
float injector_rand_f(char *iid, float var, float p);
double injector_rand_d(char *iid, double var, float p);
long double injector_rand_ld(char *iid, long double var, float p);
bool injector_rand_b(char *iid, bool var, float p);
double injector_rand_dp(char *iid, double var, float p);
int *injector_rand_ip(char *iid, int *var, float p);
float injector_rand_fp(char *iid, float var, float p);

int ret_test();
void set_func(int value);

#endif
```

```c
//Patrick Pape - 4/13 - Injector functions for injecting faults into variables in the instrumented source code

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include "injectors.h"

FILE *fp1;     //experiment list for injections and probes to use
FILE *fp2;     //variable log for injection values and probe values

int timing = 0;

//Create the list of injectors for the experiment - file is taken from simulator input

void injector_setup(){

        struct timeval time;
        gettimeofday(&time,NULL);
        srand((time.tv_sec * 1000) + (time.tv_usec / 1000));

}

//Output the bits in the variable for testing

void printBits(size_t const size, void const * const ptr)
{
        printf("size is %d - ",size);
    unsigned char *b = (unsigned char*) ptr;
    unsigned char byte;
    int i, j;

    for (i=size-1;i>=0;i--)
    {
        for (j=7;j>=0;j--)
        {
            byte = b[i] & (1<<j);
            byte >>= j;
            printf("%u", byte);
        }
        printf(" ");
    }
    puts("");
}



//Injector function to inject a fault if it is part of the experiment list
char injector_c(char *iid, char var, char inject_val, float p){

double p2 = (double)rand()/(double)RAND_MAX;

if(p2 <= p) {

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,iid)) {
```

```c
              fprintf(fp2,"%s: original = %c injection = %c\n",iid,var,inject_val);
              return inject_val; } }

fclose(fp1);
fclose(fp2);

  return var; }
else {
  return var;}
}

char *injector_s(char *iid, char *var, char *inject_val, float p){

double p2 = (double)rand()/(double)RAND_MAX;

if(p2 <= p) {

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
       if (strstr(tmp,iid)) {
              fprintf(fp2,"%s: original = %s injection = %s\n",iid,var,inject_val);
              return inject_val; } }

fclose(fp1);
fclose(fp2);

  return var; }
else {
  return var;}
}

//Fault injection function to force an integer variable fault with a given value and prob
int injector_i(char *iid, int var, int inject_val, float p){

double p2 = (double)rand()/(double)RAND_MAX;

if(p2 <= p) {

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
       if (strstr(tmp,iid)) {
              fprintf(fp2,"%s: original = %d injection = %d\n",iid,var,inject_val);
              return inject_val; } }

fclose(fp1);
fclose(fp2);

  return var; }
```

```
else {
  return var;}
}

//Fault injection function to force a short int variable fault with a given value and prob
short injector_sh(char *iid, short var, short inject_val, float p){

double p2 = (double)rand()/(double)RAND_MAX;

if(p2 <= p) {


fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,iid)) {
                fprintf(fp2,"%s: original = %d injection = %d\n",iid,var,inject_val);
                return inject_val; } }

fclose(fp1);
fclose(fp2);

  return var; }
else {
  return var;}
}

//Fault injection function to force a long int variable fault with a given value and prob
long injector_l(char *iid, long var, long inject_val, float p){

double p2 = (double)rand()/(double)RAND_MAX;

if(p2 <= p) {

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,iid)) {
                fprintf(fp2,"%s: original = %ld injection = %ld\n",iid,var,inject_val);
                return inject_val; } }

fclose(fp1);
fclose(fp2);

  return var; }
else {
  return var;}
}

//Fault injection function to force a float variable fault with a given value and prob
```

```
float injector_f(char *iid, float var, float inject_val, float p){
double p2 = (double)rand()/(double)RAND_MAX;

if(p2 <= p) {

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,iid)) {
                fprintf(fp2,"%s: original = %f injection = %f\n",iid,var,inject_val);
                return inject_val; } }

fclose(fp1);
fclose(fp2);

  return var; }
else {
  return var;}
}

//Fault injection function to force a double variable fault with a given value and prob
double injector_d(char *iid, double var, double inject_val, float p){

double p2 = (double)rand()/(double)RAND_MAX;

if(p2 <= p) {

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,iid)) {
                fprintf(fp2,"%s: original = %f injection = %f\n",iid,var,inject_val);
                return inject_val; } }

fclose(fp1);
fclose(fp2);

  return var; }
else {
  return var;}
}

//
long double injector_ld(char *iid, long double var, long double inject_val, float p){

double p2 = (double)rand()/(double)RAND_MAX;

if(p2 <= p) {

fp1 = fopen("exp_list.txt", "r");
```

```
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,iid)) {
                fprintf(fp2,"%s: original = %Lf injection = %Lf\n",iid,var,inject_val);
                return inject_val; } }

fclose(fp1);
fclose(fp2);

  return var; }
else {
  return var;}
}

//bool injector_b(char *iid, bool var, bool inject_val, float p);
//*************************************************************************************************************************************************************************


//Fault injection function to force a char variable fault given a probability and random value
char injector_rand_c(char *iid, char var, float p){

double p2 = (double)rand()/(double)RAND_MAX;
unsigned int bitCount = 8 * sizeof(char);
int fault_loc = rand()%bitCount;                   //removed the + 1, created an error when the bitplace chosen was 8 (or MAX SIZE) - wouldn't
       affect a single character
int fault_inj = (1 << fault_loc);

        unsigned long temp = *(unsigned long*)&var;
        temp = temp ^ (1 << fault_loc);
        char inject_val = *(char*)&temp;

//Test Code
/*
printf("bitCount = %d randomBitPlace = %d \n", bitCount, fault_loc);
printf("var ^ fault_inj = inject_val \n");
printBits(sizeof(char), &var);
printBits(sizeof(char), &fault_inj);
printf("-------------------------------\n");
printBits(sizeof(char), &inject_val);*/

        if(p2 <= p) {

                fp1 = fopen("exp_list.txt", "r");
                fp2 = fopen("var_log.txt", "a");

                char tmp[256] = {0x0};

                while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
                        if (strstr(tmp,iid)) {
                                fprintf(fp2,"%s: original = %c injection = %c\n",iid,var,inject_val);
                                return inject_val; } }

        fclose(fp1);
        fclose(fp2);

  return var; }
else {
  return var;}
}

//Fault injection function to force a string variable fault given a probability and random value
char *injector_rand_s(char *iid, char *var, float p){

double p2 = (double)rand()/(double)RAND_MAX;
unsigned int bitCount = 8 * strlen(var);
int fault_loc = rand()%bitCount;
int fault_inj = 0 ^ (1 << fault_loc);

int i;
unsigned long temp;
char * inject_val = var;
char var_copy[strlen(var)];
int inj_byte = fault_loc/8;
int fault_loc_byte = fault_loc%8;



//printf("inj_byte = %d \n", inj_byte);
//printf("fault_loc_byte = %d \n", fault_loc_byte);

for(i=0;i<strlen(inject_val);i++) {
        var_copy[i] = var[i];
        if(i == inj_byte) {
                temp = *(unsigned long*)&inject_val[i];
                temp = temp ^ (1 << fault_loc_byte);
                inject_val[i] = *(char*)&temp; }
}

//Test Code
/*
printf("bitCount = %d randomBitPlace = %d \n", bitCount, fault_loc);
printf("var ^ fault_inj = inject_val \n");
printBits(strlen(var), var);
printBits(strlen(var), &fault_inj);
printf("-------------------------------\n");
printBits(strlen(var), inject_val);*/

        if(p2 <= p) {

                fp1 = fopen("exp_list.txt", "r");
                fp2 = fopen("var_log.txt", "a");

                char tmp[256] = {0x0};

                while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
                        if (strstr(tmp,iid)) {
                                fprintf(fp2,"%s: original = %s injection = %s\n",iid,var_copy,inject_val);
                                return inject_val; } }

        fclose(fp1);
        fclose(fp2);

  return var; }
else {
  return var;}
}

//Fault injection function to force an int variable fault with a given prob and random bit-flip
int injector_rand_i(char *iid, int var, float p){

double p2 = (double)rand()/(double)RAND_MAX;
unsigned int bitCount = 8 * sizeof(int);
int fault_loc = rand()%bitCount;
int fault_inj = (1 << fault_loc);

int inject_val = var ^ (1 << fault_loc);

//Test Code
/*
printf("bitCount = %d randomBitPlace = %d \n", bitCount, fault_loc);
printf("var ^ fault_inj = inject_val \n");
printBits(sizeof(int), &var);
printBits(sizeof(int), &fault_inj);
printf("-------------------------------\n");
printBits(sizeof(int), &inject_val);
printf("inject_val = %d\n",inject_val);*/

        if(p2 <= p) {

        fp1 = fopen("exp_list.txt", "r");
        fp2 = fopen("var_log.txt", "a");

        char tmp[256] = {0x0};


                while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
                if (strstr(tmp,iid)) {
                        fprintf(fp2,"%s: original = %d injection = %d\n",iid,var,inject_val);
                        return inject_val; } }

        fclose(fp1);
        fclose(fp2);

          return var; }
        else {
          return var;}
        }

//Fault injection function to force a short variable fault with given probability and random value
short injector_rand_sh(char *iid, short var, float p){

double p2 = (double)rand()/(double)RAND_MAX;
unsigned int bitCount = 8 * sizeof(short);
int fault_loc = rand()%bitCount;

        unsigned long temp = *(unsigned long*)&var;
        temp = temp ^ (1 << fault_loc);
        short inject_val = *(short*)&temp;

        if(p2 <= p) {

                fp1 = fopen("exp_list.txt", "r");
                fp2 = fopen("var_log.txt", "a");

                char tmp[256] = {0x0};

                while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
                        if (strstr(tmp,iid)) {
                                fprintf(fp2,"%s: original = %d injection = %d\n",iid,var,inject_val);
                                return inject_val; } }

        fclose(fp1);
        fclose(fp2);

  return var; }
else {
  return var;}
}

//Fault injection function to force long variable fault with given probability and random value
long injector_rand_l(char *iid, long var, float p){

double p2 = (double)rand()/(double)RAND_MAX;
unsigned int bitCount = 8 * sizeof(long);
int fault_loc = rand()%bitCount;
int fault_inj = (1 << fault_loc);

        unsigned long long temp = *(unsigned long long*)&var;
        temp = temp ^ (1 << fault_loc);
        long inject_val = *(long*)&temp;

//Test Code
/*
printf("bitCount = %d randomBitPlace = %d \n", bitCount, fault_loc);
printf("var ^ fault_inj = inject_val \n");
printBits(sizeof(long), &var);
printBits(sizeof(long), &fault_inj);
printf("-------------------------------\n");
printBits(sizeof(long), &inject_val);*/

        if(p2 <= p) {

                fp1 = fopen("exp_list.txt", "r");
                fp2 = fopen("var_log.txt", "a");

                char tmp[256] = {0x0};

                while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
                        if (strstr(tmp,iid)) {
                                fprintf(fp2,"%s: original = %ld injection = %ld\n",iid,var,inject_val);
                                return inject_val; } }
        fclose(fp1);
        fclose(fp2);

  return var; }
else {
  return var;}
}


//Fault injection function to force a float variable fault with given probability and random value
float injector_rand_f(char *iid, float var, float p){

double p2 = (double)rand()/(double)RAND_MAX;
unsigned int bitCount = 8 * sizeof(float);
int fault_loc = rand()%bitCount;

        unsigned long temp = *(unsigned long*)&var;
        temp = temp ^ (1 << fault_loc);
        float inject_val = *(float*)&temp;

        if(p2 <= p) {

                fp1 = fopen("exp_list.txt", "r");
                fp2 = fopen("var_log.txt", "a");

                char tmp[256] = {0x0};

                while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
                        if (strstr(tmp,iid)) {
                                fprintf(fp2,"%s: original = %f injection = %f\n",iid,var,inject_val);
                                return inject_val; } }

        fclose(fp1);
        fclose(fp2);

  return var; }
else {
  return var;}
}

//Fault injection function to force a double variable fault with a given probability and random value
double injector_rand_d(char *iid, double var, float p){

double p2 = (double)rand()/(double)RAND_MAX;
unsigned int bitCount = 8 * sizeof(double);
int fault_loc = rand()%bitCount;
int fault_inj = (1 << fault_loc);

        unsigned long long temp = *(unsigned long long*)&var;
        temp = temp ^ (1 << fault_loc);
        double inject_val = *(double*)&temp;

//Test Code
/*
printf("bitCount = %d randomBitPlace = %d \n", bitCount, fault_loc);
printf("var ^ fault_inj = inject_val \n");
```

```
printBits(sizeof(double), &var);
printBits(sizeof(double), &fault_inj);
printf("-------------------------------\n");
printBits(sizeof(unsigned long long), &temp);*/

        if(p2 <= p) {

                fp1 = fopen("exp_list.txt", "r");
                fp2 = fopen("var_log.txt", "a");

                char tmp[256] = {0x0};

                while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
                        if (strstr(tmp,iid)) {
                                fprintf(fp2,"%s: original = %f injection = %f\n",iid,var,inject_val);
                                return inject_val; } }

        fclose(fp1);
        fclose(fp2);

  return var; }
else {
  return var;}
}

//Fault injection function to force a long double variable fault with given probability and random value
long double injector_rand_ld(char *iid, long double var, float p){

double p2 = (double)rand()/(double)RAND_MAX;
unsigned int bitCount = 8 * sizeof(long double);
int fault_loc = rand()%bitCount;
int fault_inj = (1 << fault_loc);

        unsigned long long temp = *(unsigned long long*)&var;
        temp = temp ^ (1 << fault_loc);
        long double inject_val = *(long double*)&temp;

//Test Code
/*
printf("bitCount = %d randomBitPlace = %d \n", bitCount, fault_loc);
printf("var ^ fault_inj = inject_val \n");
printBits(sizeof(long double), &var);
printBits(sizeof(long double), &fault_inj);
printf("-------------------------------\n");
printBits(sizeof(long double), &temp);*/

        if(p2 <= p) {

                fp1 = fopen("exp_list.txt", "r");
                fp2 = fopen("var_log.txt", "a");

                char tmp[256] = {0x0};

                while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
                        if (strstr(tmp,iid)) {
                                fprintf(fp2,"%s: original = %Lf injection = %Lf\n",iid,var,inject_val);
                                return inject_val; } }

        fclose(fp1);
        fclose(fp2);

  return var; }
else {
  return var;}
}

//Fault injection function to force a bool variable fault with given probability and random value
bool injector_rand_b(char *iid, bool var, float p){

double p2 = (double)rand()/(double)RAND_MAX;
unsigned int bitCount = 8 * sizeof(bool);
int fault_loc = rand()%bitCount;

        unsigned long temp = *(unsigned long*)&var;
        temp = temp ^ (1 << fault_loc);
        bool inject_val = *(bool*)&temp;

        if(p2 <= p) {

                fp1 = fopen("exp_list.txt", "r");
                fp2 = fopen("var_log.txt", "a");

                char tmp[256] = {0x0};

                while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
                        if (strstr(tmp,iid)) {
                                fprintf(fp2,"%s: original = %d injection = %d\n",iid,var,inject_val);
                                return inject_val; } }

        fclose(fp1);
        fclose(fp2);

  return var; }
else {
  return var;}
}

//Fault injection function to force a double variable fault in the address pointed to by the pointer variable
double injector_rand_dp(char *iid, double var, float p){

        double var_value = var;
        var_value = injector_rand_d(iid, var_value, p);
        var = var_value;

        return var;
}

//Fault injection function to force an integer variable fault in the address pointed to by the pointer variable
int *injector_rand_ip(char *iid, int *var, float p){

        int var_value = *var;
        var_value = injector_rand_i(iid, var_value, p);
        var = &var_value;

        return *var;
}

//Fault injection function to force a float variable fault in the address pointed to be the pointer variable
float injector_rand_fp(char *iid, float var, float p){

        float var_value = var;
        var_value = injector_rand_f(iid, var_value, p);
        var = var_value;

        return var;
}
```

```
//Patrick Pape - 4/13 - Header file for probe functions

#ifndef PROBE_H_INCLUDED
#define PROBE_H_INCLUDED

#include <stdbool.h>

//Create the list of injectors for the experiment - file is taken from simulator input
void create_probe_list();

//Probe function to read a variable if it is part of the experiment list
void probe_c(char *pid, char var_probe);
void probe_s(char *pid, char *var_probe);
void probe_i(char *pid, int var_probe);
void probe_sh(char *pid, short var_probe);
void probe_l(char *pid, long var_probe);
void probe_f(char *pid, float var_probe);
void probe_d(char *pid, double var_probe);
void probe_ld(char *pid, long double var_probe);
void probe_b(char *pid, bool var_probe);
void probe_fp(char *pid, float var_probe);
void probe_ip(char *pid, int *var_probe);
void probe_dp(char *pid, double var_probe);

#endif
```

```
//Patrick Pape - 4/13 - Probe functions for reading variable values at set locations in the code

#include "probes.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

FILE *fp1;     //experiment list for which injections and probes to use
FILE *fp2;     //variable log for injection values and probe values

//Create the list of injectors for the experiment - file is taken from simulator input

//Probe size of function
```

```
void probe_size(size_t const size, char * name) {

fp2 = fopen("var_log.txt", "a");

fprintf(fp2, "%s variable is %d bytes \n", name, size);

fclose(fp2);

}

//Probe function to read char variable
void probe_c(char *pid, char var_probe){

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};;

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,pid)) {
                fprintf(fp2,"%s => %c\n",pid,var_probe);}
 }

fclose(fp1);
fclose(fp2);
}

//Probe function to read string variable
void probe_s(char *pid, char *var_probe){

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};;

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,pid)) {
                fprintf(fp2,"%s => %s\n",pid,var_probe);}
 }

fclose(fp1);
fclose(fp2);
}

//Probe function to read integer variable
void probe_i(char *pid, int var_probe){

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};;

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,pid)) {
                fprintf(fp2,"%s => %d\n",pid,var_probe);}
 }
```

```c
fclose(fp1);
fclose(fp2);
}

//Probe function for short variables
void probe_sh(char *pid, short var_probe){

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};;

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,pid)) {
                fprintf(fp2,"%s => %d\n",pid,var_probe);}
 }

fclose(fp1);
fclose(fp2);
}

//Probe function for long variables
void probe_l(char *pid, long var_probe){

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};;

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,pid)) {
                fprintf(fp2,"%s => %ld\n",pid,var_probe);}
 }

fclose(fp1);
fclose(fp2);
}

//Probe function for float variables
void probe_f(char *pid, float var_probe){

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};;

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,pid)) {
                fprintf(fp2,"%s => %f\n",pid,var_probe);}
 }

fclose(fp1);
fclose(fp2);
}
```

```
//Probe function for double variables
void probe_d(char *pid, double var_probe){

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};;

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,pid)) {
                fprintf(fp2,"%s => %f\n",pid,var_probe);}
 }

fclose(fp1);
fclose(fp2);
}

//Probe function for long double variables
void probe_ld(char *pid, long double var_probe){

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};;

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,pid)) {
                fprintf(fp2,"%s => %Lf\n",pid,var_probe);}
 }

fclose(fp1);
fclose(fp2);
}

//Prove function for bool variables
void probe_b(char *pid, bool var_probe) {

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};;

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
        if (strstr(tmp,pid)) {
                fprintf(fp2,"%s => %d\n",pid,var_probe);}
 }

fclose(fp1);
fclose(fp2);
}

//Prove function for integer pointers variables
void probe_ip(char *pid, int *var_probe) {

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");
char tmp[256] = {0x0};;

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
       if (strstr(tmp,pid)) {
              fprintf(fp2,"%s => %d\n",pid,*var_probe);}
 }

fclose(fp1);
fclose(fp2);
}

//Prove function for float pointers variables
void probe_fp(char *pid, float var_probe) {

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};;

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
       if (strstr(tmp,pid)) {
              fprintf(fp2,"%s => %f\n",pid,var_probe);}
 }

fclose(fp1);
fclose(fp2);
}

//Prove function for double pointers variables
void probe_dp(char *pid, double var_probe) {

fp1 = fopen("exp_list.txt", "r");
fp2 = fopen("var_log.txt", "a");

char tmp[256] = {0x0};;

while(fp1 != NULL && fgets(tmp, sizeof(tmp), fp1) != NULL) {
       if (strstr(tmp,pid)) {
              fprintf(fp2,"%s => %lf\n",pid,var_probe);}
 }

fclose(fp1);
fclose(fp2);
}
```

```
//Patrick Pape - 4/13 - Program to test each of the injector functions by creating a variable and outputting the size and data in bits.
//The program then injects a bit fault into the variable and displays the new data stored in each of the variables.

#include <stdio.h>
#include <sys/types.h>
#include <getopt.h>
#include <assert.h>
#include <stdbool.h>
```

```
#include "injectors.c"
#include "probes.c"

int main() {

        //setup the fault injection framework
        injector_setup();

        //output the size and contents for each variable
        printf("creating variables...\n");

        bool x = false;
        int y = 0;
        char z[6] = "string";

        char test_c = (char)rand();
        int test_i = (int)rand();
        short test_sh = (short)rand();
        long test_l = (long)rand();
        float test_f = (float)rand();
        double test_d = (double)rand();
        long double test_ld = (long double)rand();
        bool test_b = (bool)rand();
        char * test_string = z;
        unsigned u;

        int test_i_b = (int)rand();
        float test_f_b = (float)rand();
        double test_d_b = (double)rand();

        int *ip;
        float *flp;
        double *dp;

        ip = &test_i_b;
        flp = &test_f_b;
        dp = &test_d_b;

        //test the injectors
        printf("injecting faults.....\n");

        char bef_c = test_c;
        int bef_i = test_i;
        short bef_sh = test_sh;
        long bef_l = test_l;
        float bef_f = test_f;
        double bef_d = test_d;
        long double bef_ld;
        bool bef_b = test_b;
        char * bef_string = z;
        int bef_ip = *ip;
        float bef_flp = *flp;
        double bef_dp = *dp;

        test_c = injector_rand_c("test_c", test_c, 1);
        test_i = injector_rand_i("test_i", test_i, 1);
```

```
test_sh = injector_rand_sh("test_sh", test_sh, 1);
test_l = injector_rand_l("test_l", test_l, 1);
test_f = injector_rand_f("test_f", test_f, 1);
test_d = injector_rand_d("test_d", test_d, 1);
test_ld = injector_rand_ld("test_ld", test_ld, 1);
test_b = injector_rand_b("test_b", test_b, 1);
test_string = injector_rand_s("test_s", test_string, 1);
*ip = injector_rand_ip("test_ip", ip, 1);
*flp = injector_rand_fp("test_flp", *flp, 1);
*dp = injector_rand_dp("test_dp", *dp, 1);

//output the new values for each variable after injection
printf("outputting results.....\n");

//Char variables injection

printf("\nChar\n");
printf("---------------------------------------------\n");
printf("Before: %c - ",bef_c);
printBits(sizeof(char), &bef_c);
printf("After: %c - ", test_c);
printBits(sizeof(char), &test_c);

//Int variable injection

printf("\nInt\n");
printf("---------------------------------------------\n");
printf("Before: %d - ",bef_i);
printBits(sizeof(int), &bef_i);
printf("After: %d - ", test_i);
printBits(sizeof(int), &test_i);

//Short variable injection

printf("\nShort\n");
printf("---------------------------------------------\n");
printf("Before: %d - ",bef_sh);
printBits(sizeof(short), &bef_sh);
printf("After: %d - ", test_sh);
printBits(sizeof(short), &test_sh);

//Long variable injection

printf("\nLong\n");
printf("---------------------------------------------\n");
printf("Before: %ld - ",bef_l);
printBits(sizeof(long), &bef_l);
printf("After: %ld - ", test_l);
printBits(sizeof(long), &test_l);

//Float variable injection

printf("\nFloat\n");
printf("---------------------------------------------\n");
printf("Before: %f - ",bef_f);
printBits(sizeof(float), &bef_f);
```

```
printf("After: %f - ", test_f);
printBits(sizeof(float), &test_f);

//Double variable injection

printf("\nDouble\n");
printf("---------------------------------------------\n");
printf("Before: %lf - ",bef_d);
printBits(sizeof(double), &bef_d);
printf("After: %lf - ", test_d);
printBits(sizeof(double), &test_d);

//Long double variable injection

printf("\nLong Double\n");
printf("---------------------------------------------\n");
printf("Before: %Le - ",bef_ld);
printBits(sizeof(long double), &bef_ld);
printf("After: %Le - ", test_ld);
printBits(sizeof(long double), &test_ld);

//Bool variable injection

printf("\nBool\n");
printf("---------------------------------------------\n");
printf("Before: %d - ",bef_b);
printBits(sizeof(bool), &bef_b);
printf("After: %d - ", test_b);
printBits(sizeof(bool), &test_b);

//Int pointer injection

printf("\nInt Pointer\n");
printf("---------------------------------------------\n");
printf("Before: %d - ",bef_ip);
printBits(sizeof(int), &bef_ip);
printf("After: %d - ", *ip);
printBits(sizeof(int), &(*ip));

//Float pointer injection

printf("\nFloat Pointer\n");
printf("---------------------------------------------\n");
printf("Before: %f - ",bef_flp);
printBits(sizeof(float), &bef_flp);

//Double pointer injection

printf("\nDouble Pointer\n");
printf("---------------------------------------------\n");
printf("Before: %lf - ",bef_dp);
printBits(sizeof(double), &bef_dp);
printf("After: %lf - ", *dp);

//String variable injection
```

```
        printf("\nString\n");
        printf("--------------------------------------------\n");
        printf("Before: %s - ",bef_string);
        printBits(strlen(bef_string), bef_string);
        printf("After: %s - ", test_string);
        printBits(strlen(test_string), test_string);

        return 0;
}
```

Appendix B

Sample Code Coverage Output

```
        -:    0:Source:src/mv.c
        -:    0:Graph:src/mv.gcno
        -:    0:Data:src/mv.gcda
        -:    0:Runs:1000
        -:    0:Programs:1
        -:    1:/* mv -- move or rename files
        -:    2:   Copyright (C) 1986-2014 Free Software Foundation, Inc.
        -:    3:
        -:    4:   This program is free software: you can redistribute it and/or modify
        -:    5:   it under the terms of the GNU General Public License as published by
        -:    6:   the Free Software Foundation, either version 3 of the License, or
        -:    7:   (at your option) any later version.
        -:    8:
        -:    9:   This program is distributed in the hope that it will be useful,
        -:   10:   but WITHOUT ANY WARRANTY; without even the implied warranty of
        -:   11:   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
        -:   12:   GNU General Public License for more details.
        -:   13:
        -:   14:   You should have received a copy of the GNU General Public License
        -:   15:   along with this program. If not, see <http://www.gnu.org/licenses/>. */
        -:   16:
        -:   17:/* Written by Mike Parker, David MacKenzie, and Jim Meyering */
        -:   18:
        -:   19:#include <config.h>
        -:   20:#include <stdio.h>
        -:   21:#include <getopt.h>
        -:   22:#include <sys/types.h>
        -:   23:#include <assert.h>
        -:   24:#include <selinux/selinux.h>
        -:   25:
        -:   26:#include "system.h"
        -:   27:#include "backupfile.h"
        -:   28:#include "copy.h"
        -:   29:#include "cp-hash.h"
        -:   30:#include "error.h"
        -:   31:#include "filenamecat.h"
        -:   32:#include "quote.h"
        -:   33:#include "remove.h"
        -:   34:#include "root-dev-ino.h"
        -:   35:#include "priv-set.h"
        -:   36:
        -:   37:#include "injectors.c"
        -:   38:#include "probes.c"
        -:   39:
        -:   40:/* The official name of this program (e.g., no 'g' prefix). */
```

```
    -:   41:#define PROGRAM_NAME "mv"
    -:   42:
    -:   43:#define AUTHORS \
    -:   44:  proper_name ("Mike Parker"), \
    -:   45:  proper_name ("David MacKenzie"), \
    -:   46:  proper_name ("Jim Meyering")
    -:   47:
    -:   48:/* For long options that have no equivalent short option, use a
    -:   49:   non-character as a pseudo short option, starting with CHAR_MAX + 1. */
    -:   50:enum
    -:   51:{
    -:   52:  STRIP_TRAILING_SLASHES_OPTION = CHAR_MAX + 1
    -:   53:};
    -:   54:
    -:   55:/* Remove any trailing slashes from each SOURCE argument. */
    -:   56:static bool remove_trailing_slashes;
    -:   57:
    -:   58:static struct option const long_options[] =
    -:   59:{
    -:   60:  {"backup", optional_argument, NULL, ’b’},
    -:   61:  {"context", no_argument, NULL, ’Z’},
    -:   62:  {"force", no_argument, NULL, ’f’},
    -:   63:  {"interactive", no_argument, NULL, ’i’},
    -:   64:  {"no-clobber", no_argument, NULL, ’n’},
    -:   65:  {"no-target-directory", no_argument, NULL, ’T’},
    -:   66:  {"strip-trailing-slashes", no_argument, NULL, STRIP_TRAILING_SLASHES_OPTION},
    -:   67:  {"suffix", required_argument, NULL, ’S’},
    -:   68:  {"target-directory", required_argument, NULL, ’t’},
    -:   69:  {"update", no_argument, NULL, ’u’},
    -:   70:  {"verbose", no_argument, NULL, ’v’},
    -:   71:  {GETOPT_HELP_OPTION_DECL},
    -:   72:  {GETOPT_VERSION_OPTION_DECL},
    -:   73:  {NULL, 0, NULL, 0}
    -:   74:};
    -:   75:
    -:   76:static void
 1000:   77:rm_option_init (struct rm_options *x)
    -:   78:{
 1000:   79:  x->ignore_missing_files = false;
 1000:   80:  x->remove_empty_directories = true;
 1000:   81:  x->recursive = true;
 1000:   82:  x->one_file_system = false;
    -:   83:
    -:   84:  /* Should we prompt for removal, too? No. Prompting for the ’move’
    -:   85:     part is enough. It implies removal. */
 1000:   86:  x->interactive = RMI_NEVER;
 1000:   87:  x->stdin_tty = false;
    -:   88:
 1000:   89:  x->verbose = false;
    -:   90:
    -:   91:  /* Since this program may well have to process additional command
    -:   92:     line arguments after any call to ’rm’, that function must preserve
    -:   93:     the initial working directory, in case one of those is a
    -:   94:     ’.’-relative name. */
 1000:   95:  x->require_restore_cwd = true;
    -:   96:
```

```
    -:   97:  {
    -:   98:    static struct dev_ino dev_ino_buf;
 1000:   99:    x->root_dev_ino = get_root_dev_ino (&dev_ino_buf);
 1000:  100:    if (x->root_dev_ino == NULL)
#####:  101:      error (EXIT_FAILURE, errno, _("failed to get attributes of %s"),
    -:  102:             quote ("/"));
    -:  103:  }
 1000:  104:}
    -:  105:
    -:  106:static void
 1000:  107:cp_option_init (struct cp_options *x)
    -:  108:{
 1000:  109:  bool selinux_enabled = (0 < is_selinux_enabled ());
    -:  110:
 1000:  111:  cp_options_default (x);
 1000:  112:  x->copy_as_regular = false;  /* FIXME: maybe make this an option */
 1000:  113:  x->reflink_mode = REFLINK_NEVER;
 1000:  114:  x->dereference = DEREF_NEVER;
 1000:  115:  x->unlink_dest_before_opening = false;
 1000:  116:  x->unlink_dest_after_failed_open = false;
 1000:  117:  x->hard_link = false;
 1000:  118:  x->interactive = I_UNSPECIFIED;
 1000:  119:  x->move_mode = true;
 1000:  120:  x->one_file_system = false;
 1000:  121:  x->preserve_ownership = true;
 1000:  122:  x->preserve_links = true;
 1000:  123:  x->preserve_mode = true;
 1000:  124:  x->preserve_timestamps = true;
 1000:  125:  x->explicit_no_preserve_mode= false;
 1000:  126:  x->preserve_security_context = selinux_enabled;
 1000:  127:  x->set_security_context = false;
 1000:  128:  x->reduce_diagnostics = false;
 1000:  129:  x->data_copy_required = true;
 1000:  130:  x->require_preserve = false;  /* FIXME: maybe make this an option */
 1000:  131:  x->require_preserve_context = false;
 1000:  132:  x->preserve_xattr = true;
 1000:  133:  x->require_preserve_xattr = false;
 1000:  134:  x->recursive = true;
 1000:  135:  x->sparse_mode = SPARSE_AUTO;  /* FIXME: maybe make this an option */
 1000:  136:  x->symbolic_link = false;
 1000:  137:  x->set_mode = false;
 1000:  138:  x->mode = 4544;
 1000:  139:  x->stdin_tty = isatty (STDIN_FILENO);
    -:  140:
 1000:  141:  x->open_dangling_dest_symlink = false;
 1000:  142:  x->update = false;
 1000:  143:  x->verbose = false;
 1000:  144:  x->dest_info = NULL;
 1000:  145:  x->src_info = NULL;
    -:  146:
 1000:  147:}
    -:  148:
    -:  149:/* FILE is the last operand of this command.  Return true if FILE is a
    -:  150:   directory.  But report an error if there is a problem accessing FILE, other
    -:  151:   than nonexistence (errno == ENOENT).  */
    -:  152:
    -:  153:static bool
  900:  154:target_directory_operand (char const *file)
    -:  155:{
    -:  156:  struct stat st;
  900:  157:  int err = (stat (file, &st) == 0 ? 0 : errno);
  900:  158:  bool is_a_dir = !err && S_ISDIR (st.st_mode);
  900:  159:  if (err && err != ENOENT)
#####:  160:    error (EXIT_FAILURE, err, _("failed to access %s"), quote (file));
  900:  161:  return is_a_dir;
    -:  162:}
    -:  163:
    -:  164:/* Move SOURCE onto DEST.  Handles cross-file-system moves.
    -:  165:   If SOURCE is a directory, DEST must not exist.
    -:  166:   Return true if successful.  */
    -:  167:
    -:  168:static bool
 1000:  169:do_move (const char *source, const char *dest, const struct cp_options *x)
    -:  170:{
    -:  171:  bool copy_into_self;
    -:  172:  bool rename_succeeded;
 1000:  173:  bool ok = copy (source, dest, false, x, &copy_into_self, &rename_succeeded);
    -:  174:
 1000:  175:  if (ok)
    -:  176:    {
    -:  177:      char const *dir_to_remove;
 1000:  178:      if (copy_into_self)
    -:  179:        {
    -:  180:          /* In general, when copy returns with copy_into_self set, SOURCE is
    -:  181:             the same as, or a parent of DEST.  In this case we know it’s a
    -:  182:             parent.  It doesn’t make sense to move a directory into itself, and
    -:  183:             besides in some situations doing so would give highly nonintuitive
    -:  184:             results.  Run this ’mkdir b; touch a c; mv * b’ in an empty
    -:  185:             directory.  Here’s the result of running echo $(find b -print):
    -:  186:             b b/a b/b b/b/a b/c.  Notice that only file ’a’ was copied
    -:  187:             into b/b.  Handle this by giving a diagnostic, removing the
    -:  188:             copied-into-self directory, DEST (’b/b’ in the example),
    -:  189:             and failing.  */
    -:  190:
#####:  191:          dir_to_remove = NULL;
#####:  192:          ok = false;
    -:  193:        }
 1000:  194:      else if (rename_succeeded)
    -:  195:        {
    -:  196:          /* No need to remove anything.  SOURCE was successfully
    -:  197:             renamed to DEST.  Or the user declined to rename a file.  */
 1000:  198:          dir_to_remove = NULL;
    -:  199:        }
    -:  200:      else
    -:  201:        {
    -:  202:          /* This may mean SOURCE and DEST referred to different devices.
    -:  203:             It may also conceivably mean that even though they referred
    -:  204:             to the same device, rename wasn’t implemented for that device.
    -:  205:
    -:  206:             E.g., (from Joel N. Weber),
    -:  207:             [...] there might someday be cases where you can’t rename
    -:  208:             but you can copy where the device name is the same, especially
    -: 209:             on Hurd. Consider an ftpfs with a primitive ftp server that
    -: 210:             supports uploading, downloading and deleting, but not renaming.
    -: 211:
    -: 212:             Also, note that comparing device numbers is not a reliable
    -: 213:             check for 'can-rename'. Some systems can be set up so that
    -: 214:             files from many different physical devices all have the same
    -: 215:             st_dev field. This is a feature of some NFS mounting
    -: 216:             configurations.
    -: 217:
    -: 218:             We reach this point if SOURCE has been successfully copied
    -: 219:             to DEST. Now we have to remove SOURCE.
    -: 220:
    -: 221:             This function used to resort to copying only when rename
    -: 222:             failed and set errno to EXDEV. */
    -: 223:
#####: 224:          dir_to_remove = source;
    -: 225:        }
    -: 226:
 1000: 227:      if (dir_to_remove == NULL)
    -: 228:        {
    -: 229:          struct rm_options rm_options;
    -: 230:          enum RM_status status;
    -: 231:          char const *dir[2];
    -: 232:
 1000: 233:          rm_option_init (&rm_options);
 1000: 234:          rm_options.verbose = x->verbose;
 1000: 235:          dir[0] = dir_to_remove;
 1000: 236:          dir[1] = NULL;
    -: 237:
 1000: 238:          status = rm ((void*) dir, &rm_options);
 1000: 239:          assert (VALID_STATUS (status));
 1000: 240:          if (status == RM_ERROR)
#####: 241:            ok = false;
    -: 242:        }
    -: 243:    }
    -: 244:
 1000: 245:  return ok;
    -: 246:}
    -: 247:
    -: 248:/* Move file SOURCE onto DEST. Handles the case when DEST is a directory.
    -: 249:   Treat DEST as a directory if DEST_IS_DIR.
    -: 250:   Return true if successful. */
    -: 251:
    -: 252:static bool
 1000: 253:movefile (char *source, char *dest, bool dest_is_dir,
    -: 254:          const struct cp_options *x)
    -: 255:{
    -: 256:  bool ok;
    -: 257:
    -: 258:  /* This code was introduced to handle the ambiguity in the semantics
    -: 259:     of mv that is induced by the varying semantics of the rename function.
    -: 260:     Some systems (e.g., GNU/Linux) have a rename function that honors a
    -: 261:     trailing slash, while others (like Solaris 5,6,7) have a rename
    -: 262:     function that ignores a trailing slash. I believe the GNU/Linux
    -: 263:     rename semantics are POSIX and susv2 compliant. */
    -: 264:
```

```
 1000: 265:  if (remove_trailing_slashes)
  100: 266:    strip_trailing_slashes (source);
    -: 267:
 1000: 268:  if (dest_is_dir)
    -: 269:    {
    -: 270:      /* Treat DEST as a directory; build the full filename. */
 1000: 271:      char const *src_basename = last_component (source);
 1000: 272:      char *new_dest = file_name_concat (dest, src_basename, NULL);
 1000: 273:      strip_trailing_slashes (new_dest);
 1000: 274:      ok = do_move (source, new_dest, x);
 1000: 275:      free (new_dest);
    -: 276:    }
    -: 277:  else
    -: 278:    {
#####: 279:      ok = do_move (source, dest, x);
    -: 280:    }
    -: 281:
 1000: 282:  return ok;
    -: 283:}
    -: 284:
    -: 285:void
#####: 286:usage (int status)
    -: 287:{
#####: 288:  if (status != EXIT_SUCCESS)
#####: 289:    emit_try_help ();
    -: 290:  else
    -: 291:    {
#####: 292:      printf (_("\
    -: 293:Usage: %s [OPTION]... [-T] SOURCE DEST\n\
    -: 294:  or:  %s [OPTION]... SOURCE... DIRECTORY\n\
    -: 295:  or:  %s [OPTION]... -t DIRECTORY SOURCE...\n\
    -: 296:"),
    -: 297:              program_name, program_name, program_name);
#####: 298:      fputs (_("\
    -: 299:Rename SOURCE to DEST, or move SOURCE(s) to DIRECTORY.\n\
    -: 300:"), stdout);
    -: 301:
#####: 302:      emit_mandatory_arg_note ();
    -: 303:
#####: 304:      fputs (_("\
    -: 305:      --backup[=CONTROL]       make a backup of each existing destination file\
    -: 306:\n\
    -: 307:  -b                           like --backup but does not accept an argument\n\
    -: 308:  -f, --force                  do not prompt before overwriting\n\
    -: 309:  -i, --interactive            prompt before overwrite\n\
    -: 310:  -n, --no-clobber             do not overwrite an existing file\n\
    -: 311:If you specify more than one of -i, -f, -n, only the final one takes effect.\n\
    -: 312:"), stdout);
#####: 313:      fputs (_("\
    -: 314:      --strip-trailing-slashes  remove any trailing slashes from each SOURCE\n\
    -: 315:                                 argument\n\
    -: 316:  -S, --suffix=SUFFIX          override the usual backup suffix\n\
    -: 317:"), stdout);
#####: 318:      fputs (_("\
    -: 319:  -t, --target-directory=DIRECTORY  move all SOURCE arguments into DIRECTORY\n\
    -: 320:  -T, --no-target-directory    treat DEST as a normal file\n\
    -:  321:  -u, --update                 move only when the SOURCE file is newer\n\
    -:  322:                                 than the destination file or when the\n\
    -:  323:                                 destination file is missing\n\
    -:  324:  -v, --verbose                explain what is being done\n\
    -:  325:  -Z, --context                set SELinux security context of destination\n\
    -:  326:                                 file to default type\n\
    -:  327:"), stdout);
#####:  328:      fputs (HELP_OPTION_DESCRIPTION, stdout);
#####:  329:      fputs (VERSION_OPTION_DESCRIPTION, stdout);
#####:  330:      fputs (_("\
    -:  331:\n\
    -:  332:The backup suffix is ’˜’, unless set with --suffix or SIMPLE_BACKUP_SUFFIX.\n\
    -:  333:The version control method may be selected via the --backup option or through\n\
    -:  334:the VERSION_CONTROL environment variable.  Here are the values:\n\
    -:  335:\n\
    -:  336:"), stdout);
#####:  337:      fputs (_("\
    -:  338:  none, off       never make backups (even if --backup is given)\n\
    -:  339:  numbered, t     make numbered backups\n\
    -:  340:  existing, nil   numbered if numbered backups exist, simple otherwise\n\
    -:  341:  simple, never   always make simple backups\n\
    -:  342:"), stdout);
#####:  343:      emit_ancillary_info ();
    -:  344:    }
#####:  345:  exit (status);
    -:  346:}
    -:  347:
    -:  348:int
 1000:  349:main (int argc, char **argv)
 1000:  350:{       injector_setup();
    -:  351:  int c;
    -:  352:  bool ok;
 1000:  353:  bool make_backups = false;
    -:  354:  char *backup_suffix_string;
 1000:  355:  char *version_control_string = NULL;
    -:  356:  struct cp_options x;
 1000:  357:  char *target_directory = NULL;
 1000:  358:  bool no_target_directory = false;
    -:  359:  int n_files;
    -:  360:  char **file;
 1000:  361:  bool selinux_enabled = (0 < is_selinux_enabled ());
    -:  362:
    -:  363:  initialize_main (&argc, &argv);
 1000:  364:  set_program_name (argv[0]);
 1000:  365:  setlocale (LC_ALL, "");
 1000:  366:  bindtextdomain (PACKAGE, LOCALEDIR);
 1000:  367:  textdomain (PACKAGE);
    -:  368:
 1000:  369:  atexit (close_stdin);
    -:  370:
 1000:  371:  cp_option_init (&x);
    -:  372:
    -:  373:  /* Try to disable the ability to unlink a directory.  */
 1000:  374:  priv_set_remove_linkdir ();
    -:  375:
    -:  376:  /* FIXME: consider not calling getenv for SIMPLE_BACKUP_SUFFIX unless
```

```
        -:  377:     we’ll actually use backup_suffix_string. */
     1000:  378:  backup_suffix_string = getenv ("SIMPLE_BACKUP_SUFFIX");
        -:  379:
     1000:  380:  while ((c = getopt_long (argc, argv, "bfint:uvS:TZ", long_options, NULL))
        -:  381:         != -1)
        -:  382:    {
     1000:  383:      switch (c)
        -:  384:        {
        -:  385:        case ’b’:
      100:  386:          make_backups = true;
      100:  387:          if (optarg)
    #####:  388:            version_control_string = optarg;
      100:  389:          break;
        -:  390:        case ’f’:
      200:  391:          x.interactive = I_ALWAYS_YES;
      200:  392:          break;
        -:  393:        case ’i’:
    #####:  394:          x.interactive = I_ASK_USER;
    #####:  395:          break;
        -:  396:        case ’n’:
      100:  397:          x.interactive = I_ALWAYS_NO;
      100:  398:          break;
        -:  399:        case STRIP_TRAILING_SLASHES_OPTION:
      100:  400:          remove_trailing_slashes = true;
      100:  401:          break;
        -:  402:        case ’t’:
      100:  403:          if (target_directory)
    #####:  404:            error (EXIT_FAILURE, 0, _("multiple target directories specified"));
        -:  405:          else
        -:  406:            {
        -:  407:              struct stat st;
      100:  408:              if (stat (optarg, &st) != 0)
    #####:  409:                error (EXIT_FAILURE, errno, _("failed to access %s"),
        -:  410:                       quote (optarg));
      100:  411:              if (! S_ISDIR (st.st_mode))
    #####:  412:                error (EXIT_FAILURE, 0, _("target %s is not a directory"),
        -:  413:                       quote (optarg));
        -:  414:            }
      100:  415:          target_directory = optarg;
      100:  416:          break;
        -:  417:        case ’T’:
    #####:  418:          no_target_directory = true;
    #####:  419:          break;
        -:  420:        case ’u’:
      100:  421:          x.update = true;
      100:  422:          break;
        -:  423:        case ’v’:
      100:  424:          x.verbose = true;
      100:  425:          break;
        -:  426:        case ’S’:
      100:  427:          make_backups = true;
      100:  428:          backup_suffix_string = optarg;
      100:  429:          break;
        -:  430:        case ’Z’:
        -:  431:          /* As a performance enhancement, don’t even bother trying
        -:  432:             to "restorecon" when not on an selinux-enabled kernel. */
```

```
  100:  433:         if (selinux_enabled)
    -:  434:           {
#####:  435:             x.preserve_security_context = false;
#####:  436:             x.set_security_context = true;
    -:  437:           }
  100:  438:         break;
#####:  439:       case_GETOPT_HELP_CHAR;
#####:  440:       case_GETOPT_VERSION_CHAR (PROGRAM_NAME, AUTHORS);
    -:  441:       default:
#####:  442:         usage (EXIT_FAILURE);
    -:  443:       }
    -:  444:   }
    -:  445:
 1000:  446: n_files = argc - optind;
 1000:  447: file = argv + optind;
    -:  448:
 1000:  449: if (n_files <= !target_directory)
    -:  450:   {
#####:  451:     if (n_files <= 0)
#####:  452:       error (0, 0, _("missing file operand"));
    -:  453:     else
#####:  454:       error (0, 0, _("missing destination file operand after %s"),
    -:  455:             quote (file[0]));
#####:  456:     usage (EXIT_FAILURE);
    -:  457:   }
    -:  458:
 1000:  459: if (no_target_directory)
    -:  460:   {
#####:  461:     if (target_directory)
#####:  462:       error (EXIT_FAILURE, 0,
#####:  463:             _("cannot combine --target-directory (-t) "
    -:  464:               "and --no-target-directory (-T)"));
#####:  465:     if (2 < n_files)
    -:  466:       {
#####:  467:         error (0, 0, _("extra operand %s"), quote (file[2]));
#####:  468:         usage (EXIT_FAILURE);
    -:  469:       }
    -:  470:   }
 1000:  471: else if (!target_directory)
    -:  472:   {
  900:  473:     assert (2 <= n_files);
  900:  474:     if (target_directory_operand (file[n_files - 1]))
  900:  475:       target_directory = file[--n_files];
#####:  476:     else if (2 < n_files)
#####:  477:       error (EXIT_FAILURE, 0, _("target %s is not a directory"),
#####:  478:             quote (file[n_files - 1]));
    -:  479:   }
    -:  480:
 1000:  481: if (make_backups && x.interactive == I_ALWAYS_NO)
    -:  482:   {
#####:  483:     error (0, 0,
#####:  484:           _("options --backup and --no-clobber are mutually exclusive"));
#####:  485:     usage (EXIT_FAILURE);
    -:  486:   }
    -:  487:
 1000:  488: if (backup_suffix_string)
```

```
  100: 489:   simple_backup_suffix = xstrdup (backup_suffix_string);
    -: 490:
 1100: 491: x.backup_type = (make_backups
  100: 492:                  ? xget_version (_("backup type"),
    -: 493:                                  version_control_string)
    -: 494:                  : no_backups);
    -: 495:
 1000: 496: hash_init ();
    -: 497:
 1000: 498: if (target_directory)
    -: 499:   {
    -: 500:     int i;
    -: 501:
    -: 502:     /* Initialize the hash table only if we'll need it.
    -: 503:        The problem it is used to detect can arise only if there are
    -: 504:        two or more files to move. */
 1000: 505:     if (2 <= n_files)
#####: 506:       dest_info_init (&x);
    -: 507:
 1000: 508:     ok = true;
 2000: 509:     for (i = 0; i < n_files; ++i)  {
 1000: 510:       ok &= movefile (file[i], target_directory, true, &x);}
    -: 511:   }
    -: 512: else
#####: 513:   ok = movefile (file[0], file[1], false, &x);
    -: 514:
 1000: 515: exit (ok ? EXIT_SUCCESS : EXIT_FAILURE);
    -: 516:}
```

```
    -:    0:Source:mpglibDBL/decode_i386.c
    -:    0:Graph:mpglibDBL/decode_i386.gcno
    -:    0:Data:mpglibDBL/decode_i386.gcda
    -:    0:Runs:3
    -:    0:Programs:1
    -:    1:/*
    -:    2: * Mpeg Layer-1,2,3 audio decoder
    -:    3: * ------------------------------
    -:    4: * copyright (c) 1995,1996,1997 by Michael Hipp, All rights reserved.
    -:    5: * See also 'README'
    -:    6: *
    -:    7: * slighlty optimized for machines without autoincrement/decrement.
    -:    8: * The performance is highly compiler dependend. Maybe
    -:    9: * the decode.c version for 'normal' processor may be faster
    -:   10: * even for Intel processors.
    -:   11: */
    -:   12:
    -:   13:/* $Id: decode_i386.c,v 1.3 2003/08/12 00:02:55 snelg Exp $ */
    -:   14:
    -:   15:#ifdef HAVE_CONFIG_H
    -:   16:#include <config.h>
    -:   17:#endif
    -:   18:
    -:   19:#ifdef STDC_HEADERS
    -:   20:# include <stdlib.h>
    -:   21:# include <string.h>
```

```
        -:   22:#else
        -:   23:/*# ifndef HAVE_STRCHR
        -:   24:#  define strchr index
        -:   25:#  define strrchr rindex
        -:   26:# endif
        -:   27:char *strchr (), *strrchr ();
        -:   28:*/
        -:   29:# ifndef HAVE_MEMCPY
        -:   30:#  define memcpy(d, s, n) bcopy ((s), (d), (n))
        -:   31:#  define memmove(d, s, n) bcopy ((s), (d), (n))
        -:   32:# endif
        -:   33:#endif
        -:   34:
        -:   35:#if defined(__riscos__) && defined(FPA10)
        -:   36:#include      "ymath.h"
        -:   37:#else
        -:   38:#include      <math.h>
        -:   39:#endif
        -:   40:
        -:   41:#include "decode_i386.h"
        -:   42:#include "dct64_i386.h"
        -:   43:#include "tabinit.h"
        -:   44:
        -:   45:#include "interface.h"
        -:   46:
        -:   47:#include "probes.h"  //FI variable probe functions
        -:   48:#include "injectors.h" // FI injector functions
        -:   49:#include "errhandle.h" // error handling functions
        -:   50:
        -:   51:#ifdef WITH_DMALLOC
        -:   52:#include <dmalloc.h>
        -:   53:#endif
        -:   54:
        -:   55:Float_t *lSamp;
        -:   56:Float_t *rSamp;
        -:   57:Float_t *maxSamp;
        -:   58:unsigned char maxAmpOnly;
        -:   59:
        -:   60:int procSamp;
        -:   61:
    #####:   62:int synth_1to1_mono(PMPSTR mp, real *bandPtr,int *pnt)
        -:   63:{      //printf("decode_i386.c - synth_1to1_mono\n"); //**
        -:   64:  int ret;
    #####:   65:  int pnt1 = 0;
        -:   66:
    #####:   67:  ret = synth_1to1(mp,bandPtr,0,&pnt1);
    #####:   68:  *pnt += 64;
        -:   69:
    #####:   70:  return ret;
        -:   71:}
        -:   72:
 13536144:   73:int synth_1to1(PMPSTR mp, real *bandPtr,int channel,int *pnt)
        -:   74:{//printf("decode_i386.c - synth_1to1\n"); //**
        -:   75:/* static const int step = 2; */
        -:   76:
        -:   77:  int bo;
        -:   78: Float_t *dsamp;
 13536144:   79: real mSamp = 0;
        -:   80:
        -:   81: real *b0,(*buf)[0x110];
 13536144:   82: int clip = 0;
        -:   83: int bo1;
        -:   84:
 13536144:   85:channel ^= 1 << 1;
        -:   86:
 13536144:   87: bo = mp->synth_bo;
        -:   88:
 13536144:   89: if(!channel) { printf("hey\n");
    #####:   90:   dsamp = lSamp;
    #####:   91:   bo--;
    #####:   92:   bo &= 0xf;
    #####:   93:   buf = mp->synth_buffs[0];
        -:   94: }
        -:   95: else {
 13536144:   96:       dsamp = rSamp;
 13536144:   97:   buf = mp->synth_buffs[1];
        -:   98: }
        -:   99:
 13536144:  100: if(bo & 0x1) {
 13536144:  101:   b0 = buf[0];
 13536144:  102:   bo1 = bo;
 13536144:  103:   dct64(buf[1]+((bo+1)&0xf),buf[0]+bo,bandPtr);
        -:  104:  // dct64(buf[1]+((bo+1)&0xf),buf[0]+bo,read_imp_dp(1,bandPtr,10));
        -:  105: }
        -:  106: else {
    #####:  107:   b0 = buf[1];
    #####:  108:   bo1 = bo+1;
    #####:  109:   dct64(buf[0]+bo,buf[1]+bo+1,bandPtr);
        -:  110:   //dct64(buf[0]+bo,buf[1]+bo+1,read_imp_dp(1,bandPtr,10));
        -:  111: }
        -:  112:
 13536144:  113: mp->synth_bo = bo;
        -:  114:
 13536144:  115:if (maxAmpOnly) {
        -:  116: {
        -:  117:   register int j;
 13536144:  118:   real *window = decwin + 16 - bo1;
        -:  119:
230114448:  120:   for (j=16;j;j--,b0+=0x10,window+=0x20)
        -:  121:   {
        -:  122:     real sum;
216578304:  123:     sum  = window[0x0] * b0[0x0];
216578304:  124:     sum -= window[0x1] * b0[0x1];
216578304:  125:     sum += window[0x2] * b0[0x2];
216578304:  126:     sum -= window[0x3] * b0[0x3];
216578304:  127:     sum += window[0x4] * b0[0x4];
216578304:  128:     sum -= window[0x5] * b0[0x5];
216578304:  129:     sum += window[0x6] * b0[0x6];
216578304:  130:     sum -= window[0x7] * b0[0x7];
216578304:  131:     sum += window[0x8] * b0[0x8];
216578304:  132:     sum -= window[0x9] * b0[0x9];
216578304:  133:     sum += window[0xA] * b0[0xA];
```

```
216578304:  134:    sum -= window[0xB] * b0[0xB];
216578304:  135:    sum += window[0xC] * b0[0xC];
216578304:  136:    sum -= window[0xD] * b0[0xD];
216578304:  137:    sum += window[0xE] * b0[0xE];
216578304:  138:    sum -= window[0xF] * b0[0xF];
        -:  139:
216578304:  140:        if (sum > mSamp) {
 30107229:  141:               mSamp = sum;
        -:  142:        }
186471075:  143:        else if ((-sum) > mSamp) {
 29962909:  144:               mSamp = (-sum);
        -:  145:        }
        -:  146:  }
        -:  147:
        -:  148:  {
        -:  149:    real sum;
 13536144:  150:    sum  = window[0x0] * b0[0x0];
 13536144:  151:    sum += window[0x2] * b0[0x2];
 13536144:  152:    sum += window[0x4] * b0[0x4];
 13536144:  153:    sum += window[0x6] * b0[0x6];
 13536144:  154:    sum += window[0x8] * b0[0x8];
 13536144:  155:    sum += window[0xA] * b0[0xA];
 13536144:  156:    sum += window[0xC] * b0[0xC];
 13536144:  157:    sum += window[0xE] * b0[0xE];
        -:  158:
 13536144:  159:        if (sum > mSamp) {
    #####:  160:               mSamp = sum;
        -:  161:        }
 13536144:  162:        else if ((-sum) > mSamp) {
    #####:  163:               mSamp = (-sum);
        -:  164:        }
 13536144:  165:    b0-=0x10,window-=0x20;
        -:  166:  }
 13536144:  167:  window += bo1<<1;
        -:  168:
216578304:  169:  for (j=15;j;j--,b0-=0x10,window-=0x20)
        -:  170:  {
        -:  171:    real sum;
203042160:  172:    sum = -window[-0x1] * b0[0x0];
203042160:  173:    sum -= window[-0x2] * b0[0x1];
203042160:  174:    sum -= window[-0x3] * b0[0x2];
203042160:  175:    sum -= window[-0x4] * b0[0x3];
203042160:  176:    sum -= window[-0x5] * b0[0x4];
203042160:  177:    sum -= window[-0x6] * b0[0x5];
203042160:  178:    sum -= window[-0x7] * b0[0x6];
203042160:  179:    sum -= window[-0x8] * b0[0x7];
203042160:  180:    sum -= window[-0x9] * b0[0x8];
203042160:  181:    sum -= window[-0xA] * b0[0x9];
203042160:  182:    sum -= window[-0xB] * b0[0xA];
203042160:  183:    sum -= window[-0xC] * b0[0xB];
203042160:  184:    sum -= window[-0xD] * b0[0xC];
203042160:  185:    sum -= window[-0xE] * b0[0xD];
203042160:  186:    sum -= window[-0xF] * b0[0xE];
203042160:  187:    sum -= window[-0x0] * b0[0xF];
        -:  188:
203042160:  189:        if (sum > mSamp) {
 63129248:  190:                 mSamp = sum;
        -:  191:         }
139912912:  192:         else if ((-sum) > mSamp) {
 62933519:  193:                 mSamp = (-sum);
        -:  194:         }
        -:  195:   }
        -:  196: }
        -:  197:}
        -:  198:else {
        -:  199:
        -:  200:
        -:  201: {
        -:  202:   register int j;
    #####:  203:   real *window = decwin + 16 - bo1;
        -:  204:
    #####:  205:   for (j=16;j;j--,b0+=0x10,window+=0x20)
        -:  206:   {
        -:  207:     real sum;
    #####:  208:     sum  = window[0x0] * b0[0x0];
    #####:  209:     sum -= window[0x1] * b0[0x1];
    #####:  210:     sum += window[0x2] * b0[0x2];
    #####:  211:     sum -= window[0x3] * b0[0x3];
    #####:  212:     sum += window[0x4] * b0[0x4];
    #####:  213:     sum -= window[0x5] * b0[0x5];
    #####:  214:     sum += window[0x6] * b0[0x6];
    #####:  215:     sum -= window[0x7] * b0[0x7];
    #####:  216:     sum += window[0x8] * b0[0x8];
    #####:  217:     sum -= window[0x9] * b0[0x9];
    #####:  218:     sum += window[0xA] * b0[0xA];
    #####:  219:     sum -= window[0xB] * b0[0xB];
    #####:  220:     sum += window[0xC] * b0[0xC];
    #####:  221:     sum -= window[0xD] * b0[0xD];
    #####:  222:     sum += window[0xE] * b0[0xE];
    #####:  223:     sum -= window[0xF] * b0[0xF];
        -:  224:
    #####:  225:         *dsamp++ = (Float_t)sum;
    #####:  226:         procSamp++;
    #####:  227:         if (sum > mSamp) {
    #####:  228:                 mSamp = sum;
        -:  229:         }
    #####:  230:         else if ((-sum) > mSamp) {
    #####:  231:                 mSamp = (-sum);
        -:  232:         }
        -:  233:   }
        -:  234:
        -:  235:   {
        -:  236:     real sum;
    #####:  237:     sum  = window[0x0] * b0[0x0];
    #####:  238:     sum += window[0x2] * b0[0x2];
    #####:  239:     sum += window[0x4] * b0[0x4];
    #####:  240:     sum += window[0x6] * b0[0x6];
    #####:  241:     sum += window[0x8] * b0[0x8];
    #####:  242:     sum += window[0xA] * b0[0xA];
    #####:  243:     sum += window[0xC] * b0[0xC];
    #####:  244:     sum += window[0xE] * b0[0xE];
    #####:  245:         *dsamp++ = (Float_t)sum;
```

```
   #####: 246:         procSamp++;
   #####: 247:         if (sum > mSamp) {
   #####: 248:                 mSamp = sum;
       -: 249:         }
   #####: 250:         else if ((-sum) > mSamp) {
   #####: 251:                 mSamp = (-sum);
       -: 252:         }
   #####: 253:     b0-=0x10,window-=0x20;
       -: 254:   }
   #####: 255:   window += bo1<<1;
       -: 256:
   #####: 257:   for (j=15;j;j--,b0-=0x10,window-=0x20)
       -: 258:   {
       -: 259:     real sum;
   #####: 260:     sum = -window[-0x1] * b0[0x0];
   #####: 261:     sum -= window[-0x2] * b0[0x1];
   #####: 262:     sum -= window[-0x3] * b0[0x2];
   #####: 263:     sum -= window[-0x4] * b0[0x3];
   #####: 264:     sum -= window[-0x5] * b0[0x4];
   #####: 265:     sum -= window[-0x6] * b0[0x5];
   #####: 266:     sum -= window[-0x7] * b0[0x6];
   #####: 267:     sum -= window[-0x8] * b0[0x7];
   #####: 268:     sum -= window[-0x9] * b0[0x8];
   #####: 269:     sum -= window[-0xA] * b0[0x9];
   #####: 270:     sum -= window[-0xB] * b0[0xA];
   #####: 271:     sum -= window[-0xC] * b0[0xB];
   #####: 272:     sum -= window[-0xD] * b0[0xC];
   #####: 273:     sum -= window[-0xE] * b0[0xD];
   #####: 274:     sum -= window[-0xF] * b0[0xE];
   #####: 275:     sum -= window[-0x0] * b0[0xF];
       -: 276:
   #####: 277:         *dsamp++ = (Float_t)sum;
   #####: 278:         procSamp++;
   #####: 279:         if (sum > mSamp) {
   #####: 280:                 mSamp = sum;
       -: 281:         }
   #####: 282:         else if ((-sum) > mSamp) {
   #####: 283:                 mSamp = (-sum);
       -: 284:         }
       -: 285:   }
       -: 286: }
       -: 287:}
13536144: 288: *pnt += 128;
       -: 289:
13536144: 290: if ((Float_t)mSamp > *maxSamp)
    1481: 291:         *maxSamp = mSamp;
       -: 292:
13536144: 293: if (!channel) lSamp = dsamp;
13536144: 294: else rSamp = dsamp;
       -: 295:
13536144: 296: return clip;
       -: 297:}
       -: 298:
```