A Methodology for Increasing the Dependability of Open Source Software Components

by

Patrick Pape

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 5, 2013

Keywords: fault injection, data-flow analysis, dependability, open source

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

Abstract

With the increasing use of open source software components in real world systems, there has arisen a need for increasing the dependability of these components. The dependability of these components can be increased like any other software component, by adding error detection and error recovery mechanisms to the code to deal with unforeseen errors and faults occurring in the system that will be running the software. The major issue is not in the implementation of these error handling mechanisms, but instead in the speed that they can be applied and where to apply them. The reason for considering an open source solution to a problem is most likely to save time and money using a solution that may already do what needs to be done. The goal is to incorporate these open source components into a system after verifying and if need be, increasing the dependability of the component. But, with open source components, they are already in a maintenance stage of development, where the code is either refined or new functionality is possibly added. Thus, standard techniques that rely on use during development will take too long to implement. This thesis proposes a novel way of determining the highest priority modules in the system and the most important locations for placing error handling mechanisms in the functions within those modules.

Acknowledgments

I would like to thank my committee members for their patience and guidance throughout the process of completing this thesis. Particular thanks goes out to Dr. Hamilton who was instrumental in helping me to finalize my topic for this thesis and also broadening my view as to the possible implications of the research and potential areas for future work. Also, my thanks go out to my colleagues in the Auburn Cyber Research Center 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

List of Abbreviations

CotS	Consumer-Off-the-Shelf
CotSD	Consumer-off-the-shelf Development
OSS	Open Source Software
OSSD	Open Source Software Development

Chapter 1

Introduction

1.1 Overview

The use of open source software is becoming more prevalent in real-world projects and software development [34]. There are obvious benefits to using open source software that is readily available, including great savings in both time and money. Many organizations have turned to using open source components in their projects and for replacing more expensive consumer off-the-shelf, CotS, products or in house development. But, open source software is still limited in its applications in many real-world projects that demand a certain level of security or dependability in their software before they can be used. The issue with open source software is that there is usually no guarantee of the dependability of the components and there is often just a single developer for the system.

It is reasonable to assume that an organization that wishes to use an open source software, OSS, component would do some stress testing or other verification to determine if the component is worth using. But, the problem with post-development testing of software system is the large time and cost investment, especially when the component was developed by an outside source and no one necessarily has experience using it. Testing early and using certain standards during development can be done to avoid this. There is research available for defining key locations for error detection and recovery mechanisms or ways to ensure software dependability, as discussed in section 1.2.2 and section 1.2.4. The issue with this research is that it is intended to be used early in the development phase and not on the final product.

Consider this situation: a project developing a UAV to reconnoiter a hostile environment finds an open source solution to the ground station software for controlling and receiving

data from the UAV. The open source software was completed by a single developer and the developers of the UAV project determine that the OSS does not meet the criteria for dependability when dealing with mission-critical components. This means that the project will need to look for another alternative, such as acquiring CotS or developing the ground station software in house, costing time and money. The methodology proposed in this thesis is for increasing the dependability of OSS components so that they can be used in projects that have higher standards for dependability. A major hurdle to overcome is to create a methodology that is lightweight enough that the benefits of using OSS components remain, but the dependability of the software is increased.

This methodology utilizes fault injection testing, data-flow analysis and lightweight error detection and recovery mechanisms for increasing software dependability. A fault injection framework was written in order to obtain the values of various metrics throughout the system. The framework was modeled in part after PROPANE, a propagation analysis environment that utilizes fault injection. PROPANE allows for the user to define and instrument fault injection and sampling locations throughout the system and then outputs logs of the tests. These logs have applicable metrics for measuring the dependability of the system, such as failure rates, coverage and latency of the error detection and recovery mechanisms. PROPANE was a project completed by the DEEDs, dependable, embedded systems and software, research group [3]. The tool is outdated after not being updated since 2006. In order to leave room for the potential of future work and expanding the capabilities of the methodology, a new framework was written.

For the proposed methodology, the chosen use cases should be indicative of the real world intended use of the system. Because the testing is to be done late stage, there most likely is not time to fully test every capability of the system. The target source code is instrumented in order to determine the frequency that modules and the functions within those modules are called during the intended use cases. Once the call frequency for the modules is determined, modules with outlying frequencies, such as zero or a small fraction of calls compared to

other modules, are disqualified from being priority modules. The keys to determining the highest priority modules are: the probability that a fault occurring in memory belongs to space related to a particular function, the probability that the now corrupted code will run during the current execution of the program, and the number of functions that a corrupted variable could pass its value. In this thesis, it is assumed that the chance for a single bit fault to affect a function is equal for all functions. The important factors to determine priority are the probability that the corrupted code will be executed and the propagation. The probability of the code being executed is determined by the fraction of effective lines of code for that function and the fraction of functions that the function either sends data to or obtains data from. With this calculation completed for each function, the priority modules can be determined by the sum of the priorities of the functions within them.

A critical path is defined as the data flow paths from the highest priority function in a high priority module. All the variables in the modules are considered for error handling, and any variables in the connecting functions that have an effect on one of the variables in the module are also considered. Once these variables for a critical path are found, a golden run, a single perfect execution of the test case, is done to simulate a perfect instance of the system and the results are compared to faulty runs. Once the results of the testing are collected, the importance of each variable is calculated. Only the variables with the highest importance with respect to other variables in the critical path are considered as locations for placing the error handling mechanisms. If error handling were added to the entire system, then the overhead of the system would be too high and the time needed to implement these mechanisms would defeat the purpose of using OSS components.

Lastly, lightweight error handling mechanisms are implanted into the code to increase it's dependability. The goal for the mechanisms is to be unobtrusive and create as little overhead as possible. The error handling mechanisms to be used have already been proven to be both lightweight and effective at reducing the failure rate of variables, while at the same time maintaining a relatively low latency increase and memory usage overhead. The

novelty of this approach is in the speed at which these priority locations for placing error handling code are identified and in the accuracy in which they are identified, not in the error handling itself.

Ultimately, the goal of this research is to fill the need for a means of increasing dependability of late stage and post development OSS components so that they can be used in projects that demand higher dependability software. This research is important because there is a vast amount of OSS that could be used, but is overlooked because it cannot be trusted. The methodology is int an initial stage and will ideally be refined and expanded to include other major factors, such as security validation.

1.2 Literature Review

For this thesis I reviewed research in several areas including the use of open source software development, OSSD, in real-world projects versus the use of consumer off-the-shelf software development, CotSD, software dependability measures and testing; fault injection techniques and methodologies; fault detection and recovery mechanisms; data flow analysis; and wrapper-based software implementations used increasing software dependability. In this section of the thesis, I present the references used along with a brief explanation of what they are about and how they are useful to the proposed research.

1.2.1 OSSD and CotSD

A comparison of the benefits of OSSD and CotSD in [27] states that the benefits of using OSSD over CotSD for a project are parallel and repeated development techniques; free user participants; huge development communities; and effective user testing. The paper illustrates the key differences in using the two types of software development. The portion that contributes most to this research has to do with the limitations of OSSD. These key problems include decreased communication between developers, informal management policies,

possibility of low contributors, and economic concerns. From a business point of view, developers are not being paid for their OSSD and it is possible that they will not be motivated to continue updating the software. Also, there is a higher short-term cost with integrating OSS within a system, and a lack of defined support means more time spent troubleshooting. Lack of defined support means that immediate corrections may go unanswered.

An article by Nagy, Yassin, and Bhattacherjee [28] states the benefits of using OSSD are clear: cost savings, vendor independence, and open standards. But, the limitations of OSSD are of greater importance to this research. The paper described five barriers for the integration of open source software in a real-world business environment and possible remedies. The most applicable of these barriers is the perception of technological immaturity. It is a common problem that open source solutions are often thought to be of lesser quality due to zero cost and lack of formal direction during development. This issue can be remedied with the introduction of a methodology to ensure an OSS solution with a high level of dependability. Also, the paper gave some examples of real-world businesses switching to open source software and the savings that resulted. For example, Amazon cut technology costs from 71 million to 54 million by switching to open source applications. Many companies, such as Sabre Holdings, saved tens of millions by switching to MySQL for their databases.

A paper by Kropp, Koopman, and Siewiorek [25] discusses the the creation of a set of black-box processes to certify the suitability of CotS components. There is a drawback to these black-box processes because such testing can fail to exercise significant portions of the code. The methodology requires the buyers of the CotS software to purchase oracles in order to test the code against what they want it to do, because the buyer cannot be sure of how rigorous the testing actually was.

Kropp, Koopman, and Siewiorek [25] wrote a paper about using automated testing in order to assess component robustness of CotS solutions, because even though there are savings in cost and time, the CotS component may not have been designed for robustness. Here, the robustness of the component is defined as the degree to which the component

functions correctly in the presence of exceptional inputs or stressful environmental conditions [25]. Fault injection testing is used in the testing phase to attempt to cause failures in the system under different circumstances.

The article by Ghosh and Voas [26] about software inoculation discusses the idea of using fault injection testing to identify vulnerable parts of a software system and introducing mechanisms to increase the resistance to these faults. Two solutions are offered after the testing: sending results to the developer to fix the problem at the source, or using a software wrapper to harden its defenses. Getting the vendor to make the necessary changes can be difficult, so the ideal solution is to use the software wrapper. In this case, the wrapper is useful because of the lack of access to the source code. Whenever the software is run, so is the wrapper code that is used to catch operating system exceptions and return a specified error code.

1.2.2 Software Dependability

A lightweight code analysis is suggested to evaluate dependability cases in [14]. The approach is to seek a balance between analysis and design. This is done in order to try and reduce the critical path for fulfilling critical properties, so that a simpler analysis tool can be used to establish them. These critical properties are tested using dependability cases which present explicit arguments that a system satisfies a critical property. The implementation of lightweight coding techniques through flow- and context-sensitive static analysis and the use of a single pathway on the control-flow mapping to reduce overhead are derived from the research found in this section.

The importance of variables in dependable software is the major focus in [13]. This paper is key in the research proposed in this thesis. Here, Leeke and Jhumka propose a novel metric for measuring dependable software systems. Detecting and correcting erroneous states can be very difficult. The key to this research is finding a way to determine if the correct values are held by variables determined to be important. The importance of variables is

measured through a combination of other metrics, including failure rate, spatial impact, and temporal impact. Spatial and temporal impact are metrics that measure how many adjacent modules are affected when a variable in a component is corrupted and how long the state of the program remains affected by the corrupted variable, respectively. Importance is then used to identify the most important variables, where error detection and recovery mechanisms can be focused.

The location and generation of detectors are considered in [8] and [9]. Leeke and Jhumka explore the early identification of detector locations in order to cut down on the costs of late lifecycle assessment of dependability. Insight into the potential for errors during development can help to reduce the costs as opposed to finding such errors late in development. Spatial impact is used in order to define locations of detectors because of its measure of areas of high error propagation and module coupling. Fault injection testing was done to verify the connection between late lifecyle metric spatial impact and the early detection metric of module coupling.

Leeke, Jhumka, Arif, and Anand propose a methodology for generating efficient error detection mechanisms in [9] that utilizes fault injection testing and data mining. First, fault injection testing is done on the target systems with PROPANE and the results are put through pre-processing algorithms to counter the inherit weaknesses of analyzing trends in fault injection data. After this, a symbolic pattern-learning algorithm is used to look for first-level predicates, meaning values that have an effect on the flow of the program. The algorithm uses a decision tree to create a baseline for the predicates and is later refined in order to create an efficient predicate for error detection and recovery mechanisms. This means that for each boolean value or if-type statement, there is a node in the true for whether the program should be true or false with a correct run. If the path on the decision tree for a run differs from the path for a correct run, then an error has occurred. This is done variously through over and under sampling of the data in the pre-processing phase, or other parameters.

1.2.3 Fault Injection

The next section of literature is a review of fault injection techniques and tools from past and current research that is included in order to provide insight into the design of the testing phases of the methodology to be proposed in the thesis. The first of these is [3] by Hiller, Jhumka, and Suri for the propagation analysis environment PROPANE. It is an environment that helps in the process of designing efficient error detection mechanisms by providing analysis of the effects of faults in a system. PROPANE supports both faults, by mutation of source code, and data errors, by manipulating variables and memory contents. The environment logs the results of these faults and errors and places them in a file that can be used to see the results of the test represented in various metrics. It is a portable and lightweight solution for dependability testing in a wide array of systems. But, the environment has not been updated in over eight years, so a different framework, based partially on PROPANE's functionality, was utilized for this methodology.

Chen, Tsai and Iyer [4] present hardware and software fault injection environments and injection and implementation methods. Discussion on compile-time, where faults are injected into the source code before the program image is loaded and executed in order to emulate various error types, was compared to run-time injection. Run-time injection is detailed, including several examples. Of particular interest is code insertion, such as the probes used by PROPANE, because this is the method for fault injection that is used in the proposed methodology. Lastly, the limitations of software fault injection testing compared to hardware testing are examined along with tools for completing both. The limitations indicate that hardware testing costs more and has a higher risk of damage, whereas software testing his more controllable and is easily repeatable.

A paper on enhancing fault injection testbenches by Sosnowski, Gawkowski, Zygulski and Tymoczko [5] deals with the problems of classical fault injection tools with respect to improving the experiment effectiveness and result analysis capabilities. The results of this research aided in preparing the test phases to be used in the proposed methodology and

the resulting analysis. The research is demonstrated through distribution of fault injection processes within a computer network utilizing data mining methods to collect simulation results. This resulted in increased speed in the testing phase, but was not without limitations. The golden run of the software could not be distributed across computers in the network, but the research provides insight into finding relations and critical dependability points. This means that the process for fault injection across a network, in this paper, is not applicable to the proposed methodology, but the steps for finding relations and important locations in the code are.

Moraes, Duraes, Barbosa, Martins, and Madeira [6] use software fault injection for experimental risk assessment and comparisons in. This research focuses on estimating the risk of using using outside software, such as CotS, in component-based software development. The paper presents an approach to assessing the risk of using a software component, either CotS or not. This is done through fault injection of realistic faults and their impact on possible component failures and using software complexity metrics to estimate the possibility of residual defects. The paper presents findings on determining cost of failure measurements through fault injection and frequency of faults. The findings presented were then used in deciding how best to calculate the probability of execution for faults.

A 2011 article by Natella, Cotroneo, Duraes and Madeira [7] and a dissertation by Natella [22] present an experimental study on the representativeness of injected faults on residual software faults, meaning faults that are actually experienced in the field. The study shows that the representativeness of the faults is affected by the location of the fault in the system, which results in different distributions of faults across the system. An approach is proposed to refine the fault load by removing faults that are not representative of the residual software faults. This is done to ensure that the results of the testing are meaningful and reduce the number of faults needed to test a system. The papers present several approaches to software fault injection and tools that utilize software insertion fault injection methods. This is very important to the methodology proposed in this thesis, because the information

in these papers was used to influence how to calculate the probabilities that the faults would occur in a function. This value is then used to calculate the primary metric for ranking variables.

A 2010 paper on assessing and improving the effectiveness of logs when analyzing software faults. A paper by Cinque, Cotroneo, Natella, and Pecchia [10] presents an approach to assess the effectiveness of using logs to keep track of software faults in the field, using fault injection. The use of logs is crucial to the fault injection testing method in order to collect data about the results of the testing and this research provides a means to improve the effectiveness of the logs. A fault injection framework is proposed utilizing G-SWFIT, which is used in several of the previous papers, to define a set of fault operators and generate the logs. G-SWFIT injects faults by changing the binary code corresponding to programming mistakes in high-level source code, which is primarily used when testing CotS software. The logs are then collected and refined in order to increase their efficiency. Of interest here, are the results of the number of injections and the failures that occurred from those injections that were actually logged. The data shows that before refining the logs, the number of faults logged versus unlogged were only around 20% to 30%. This shows the limitations of using logs to gather data on the software faults and their approach to increasing the percent coverage of the logs. This data influenced the method used to collect the information from the fault injection test run in this paper.

An older article by Clark and Pradhan [11] on using fault injection as a method for validating the dependability of a computer system provides details on various fault models and injection methods, as well as measures of software dependability. The research also includes information on several prevalent fault injection tools, such as MESSALINE and FIAT, that helps in looking for which fault injection tool would best suit the needs of the proposed methodology. This paper helped lead to developing a framework for the methodology, as opposed to just using MESSALINE or FIAT, which did not fill the requirements set for the methodology.

Madeira, Costa and Vieira present a paper on using software fault injection to emulate software faults [12]. The experimentation results were found by comparing real software faults found in various programs with the faults injection using the Xception tool, a SWIFI tool. The results show what kinds of faults that are possible to find and which types of faults are unlikely to be detected using this method of fault injection testing. An approach using software metrics to guide the injection process is presented when field data is not available, which is the case when trying to determine whether to integrate open source software with a system is the goal. The concepts presented in this paper were used to see what types of faults would be practical to test for using software implemented fault injection.

A 2011 paper on fault injection methodology and tools by Song, Qin, Pan, and Deng [17] describes the concept and principals of fault injection. The paper includes a detailed review of some techniques and their tools using: hardware fault injection, simulation fault injection, and software fault injection. Of particular interest is software fault injection data for compile and run-time fault injection and some useful tools, such as Ferrari, Doctor and Xception. Also, a comparison of the various tools available between each type of fault injection is presented, which aids the proposed research by helping to identify the type of fault injection tool that may need to be used. When selecting which fault injection framework to model the fault injection step of the methodology after, the data in this research helped to narrow down the choices.

Two more papers about fault injection methodologies and tools [18] and [19] present some applications of dependability validation tools. Confidence in the error handling mechanisms in a software component are analyzed through fault injection testing with MESSALINE in [18] and FIAT in [19]. Each paper provides insight into the operation and possible usefulness of their respective tools in the proposed research for the thesis. Of concern here, is not only the individual fault, but it's overall effect on the target system. Other fault injection frameworks were considered for use in the proposed methodology and were used to guide the development of the fault injection framework designed for this thesis.

A 2011 paper by Benso and Carlo on the art of fault injection [20] provides some interesting points of interest to consider when conducting any type of fault injection testing. The paper highlights good and bad practices when setting up a fault injection experiment and common errors in methodologies that affect the coherency and meaningfulness of the results of the experiments. The paper reads as a sort of guide for creating and setting up fault injection experiments. The major points include, the structure of the environment, choice of fault model, fault locations and generation of the actual target fault list. Of key importance is using this paper to ensure that the mistakes of past research are not repeated in the proposed methodology and that the results will be meaningful. The fault injection framework designed for this methodology used this paper as a guide for determining the specifics of injection location and choosing a fault model.

A paper by Jiang, Munawar, Reidemeister and Ward [21] presents a method for automatic fault detection and diagnosis using information-theoretic monitoring. The methodology involves combining two algorithms. The first is RatioScore, which is based on Jaccard coefficients, and the second is SigScore, which uses knowledge of the component dependencies. The approach monitors the state of the system and flags an anomalous state to be determined as either faults or not. The approach is limited by the availability of component dependency information and the dynamic changing of dependencies. The algorithms presented in this paper were considered for finding the dependencies of the relevant variables to be tested in the proposed methodology.

Of concern when utilizing fault injection testing is how the results can be used to identify the impact of the faults. This is often done utilizing golden runs of the system, meaning a perfect run of the system with no errors. A 2009 paper by Leeke and Jhumka [23] evaluates the use of these reference run models in fault injection analysis. Of particular interest in this paper is the finding that certain systems are not properly represented in testing with the use of these reference runs. The paper defines various types of oracles that are used to determine the impact of fault injection, such as specifications, error detection mechanisms

or golden runs, and shows that of these, golden runs are not effective when used on a system with a main control loop with an irregular period. The paper goes on to suggest a model for refining the golden run oracle for use with these types of systems. The data from this paper was useful for selecting which open source software to test the methodology on because a golden run oracle is used to look for failures during fault injection.

1.2.4 Fault Detection and Correction

There are several approaches to handling faults that may appear in software, the following papers detail techniques and approaches for handling the detection of these faults. A paper by Arora and Kulkarni [2] on error detectors and correctors details a theory for detectors and correctors that characterizes their role in achieving various types of fault-tolerances. The paper presents a large number of definitions and background on faults and the mechanisms to handle them. Also included in the research are applications of different mechanisms and what types of fault-tolerance are best fulfilled with which types of mechanisms. Examples are given for the composition of the mechanism and how they can be used in the software.

A 2009 paper on using software invariants for dynamic detection of transient errors [15] by Lisboa, Grando, Moreira and Carro discusses the possibility of using software invariants as a low cost means to detect soft errors after the execution of an algorithm, by looking for changes in the invariants for the algorithm. Usually the use of software invariants is during the development phase to check for correctness because of the huge memory footprint and overhead with late lifecycle use. The paper gives insight into a lightweight approach to hardening an algorithm without changing the algorithm itself; the overhead and computation cost of the algorithm is much smaller than duplicating an algorithm. Fault injection is used to test the algorithms and the proposed hardening methods. A software invariant is a program property that must be preserved when the code is modified, such as preconditions, post conditions or loop invariants. The focus of the methodology is the detection of single

event transient faults during the execution of the software. This paper was considered as a potential solution for detecting errors that occur in the system after fault injection.

Another 2009 article by Hamill and Goseva-Popstojanova [16] on common trends in software fault and failure data, examines fault and failure data from two real-world case studies, giving valuable data about the origins of errors in a system. In particular, the article focuses on localization of faults leading to individual software failures and the distribution of different types of software faults. The paper reveals requirement faults, coding faults and data problems as the primary types of software faults. Also, it is shown that individual software failures are likely caused by several faults spread throughout the software. The article explores whether or not certain software faults are more common than others, and how the localization of faults changes depending on the individual software failure. Most importantly, the research reveals that the trends are likely intrinsic characteristics of software faults and failure rather than being specific to a project. This means that it should be possible to create a methodology that is capable of adapting to any type of project it is used on.

1.2.5 Data-Flow Analysis

Tan and Xiong [29] published a 2011 paper on data flow error recovery using checkpointing and instruction-level fault tolerance. This paper is particularly useful because of its use of both data flow analysis for error recovery while considering fault tolerance of the software. The paper proposes an approach to data flow error recovery using checkpointing. The software is examined at the instruction level and split into protected code and unprotected code. In the protected code, data values are replicated and at certain locations, such as branch instructions, the values are compared and if there is a discrepancy then an error has occurred and a previous state of the software is loaded. This method has several similarities to the error handling mechanisms in [1].

An article by Taylor and Osterweil [30] discusses using state data flow analysis to detect anomalies in concurrent software. The article presents algorithms for using data flow analysis

for detection of variable usage errors in single and concurrent process software. The focus of the flow analysis is on variables that are shared between the multiple processes in a software system. The fundamental use of data flow analysis in this research served as good background information for deciding how best to approach the data flow analysis found in this thesis.

Bergeretti and Carre present a paper on information flow and data flow analysis specifically for while procedures[31]. In this paper, information flow is expanded from its common use for ensuring that program variables are not violating security requirements. Information flow relations are used to identify program statements that cause information to transmit between certain inputs, internal, or output values. This is useful in testing and updating programs and can extend the types of errors that can be detected automatically during static analysis. The concepts presented in this paper for handling information flow with while statements influenced the method used to determine the relevant variables in the critical path.

1.2.6 Wrapper-Based Solutions

The following two programs implement wrapper-based solutions for error detection in complex software systems. The first paper details a framework for detecting these errors [32], where the goal is to cope with software failures, based on data obtained on the application and OS levels. The framework is implemented as a wrapper about the complex system in order to detect errors. The reasoning behind the framework is that there are complex interdependencies between components and in turn these lead to subtle faults that are cause by complex triggering patterns which can escape testing. So, in order to not track down each of these complex patterns, the entire system is observed to look for errors in both application and OS activity.

The second paper [1], by Leeke and Jhumka, proposes an automated wrapper-based approach for designing dependable software. Two major activities for developing dependable software are detailed: design of the dependability mechanisms and their location. They use

their importance metric defined in an earlier paper to identify variables to be replicated. These replicated variables are used in standard, efficient dependability mechanisms, and are deployed wherever the variable is located in the software. Whenever the variable is written to, copies are made based on the level of importance of the variable. When the variable is read, the variable values are compared and the majority voting rule is used to select which of the values will be used. This means that even if one of the instances of the variables being stored is corrupted, the other two instances will have the correct value and the correct value will be returned. These algorithms were used in order to create a lightweight implementation as an alternative to the high overhead of code replication and N-version programming. The testing is done strictly through fault injection, including the determination of important variables.

1.3 Problem Statement

The use of OSS components has been used to save organizations money that can be allocated elsewhere, in addition to the time that does not need to be spent on the development of the software. This use of OSS for certain components of a system could be even higher, allowing for an even more widespread use of OSS components. The issue with the widespread use of OSS components is that most real-world projects have certain standards and software dependability requirements that need to be met before the open source solutions can be used.

Most of the research being done in current software dependability for systems where the source code is available is directed at the development process. These techniques and tools are meant to be used early in development and followed closely throughout. This is due to the fact that the later these error detection and recovery mechanisms need to be added, the more time it takes to develop and troubleshoot the changes to the code. With the case of open source software, like that of CotS software, the component being used is passed the development phase and is in its released form. The benefits of saved time and money start

to seem less appealing if a full suite of testing and possible development will need to be done in order to bring the component up to a level of dependability that meets requirements. The increase in overhead from adding these error handling mechanisms could render the OSS component ineligible for use in the system regardless of time spent testing and developing code.

There are many cases where an OSS solution could be implemented over a CotS solution. For example, there could be OSS systems available that fulfill the required performance measures and no CotS systems. Also, with many CotS systems there are additional costs, such as purchasing a license to use the software. Developing code in house or contracting an organization to develop software for a particular function is also a significant investment on a project's budget, when compared to free OSS. With CotS software or contracted development, there is a certain level of assurance that the software will be developed and tested according to the desired requirements of the customer. An issue with OSS systems is that in many cases, there is no guarantee of any standard of development or formal testing. Often, development is done by a small group or even just a single person and testing is done only loosely to ensure that the product will run and the burden of testing and troubleshooting falls to the community.

In order to solve this problem, this thesis proposes a methodology for increasing the dependability of OSS components. The focus of the research is to implement a methodology for identifying critical artifacts and locations in open source systems and adding lightweight error handling mechanisms to detect and recover from errors. There has been significant research done in the realm of fault injection, including the definition of metrics used to identify important variables and injection and sampling methods. The major issue with the importance metric defined by [13], is that it is irrelevant when considering more than one module. A new modified importance metric is proposed that takes into account the fault chance of each function in the system to create a more relevant metric, measured as the relative importance of a set of variables given a specific set of use cases. This research is

confined to specific subsets of system execution. Errors are measured according to these pre-defined runs, as it is with all fault injection research. Currently, the greatest measure of the importance of a variable is to use several metrics, including failure rate and spacial and temporal impact, described above. This methodology for determining the most important variables and their propagation is done completely through fault injection, which does not necessarily reflect real world situations that may occur while the software is in use.

In an attempt to both further verify this method of using a variable's importance to direct the use of error handling mechanisms and to explore the possibility of increasing the benefits, both fault injection and data-flow analysis is used. To limit the negative effects innate to using fault injection testing, data-flow analysis is done on modules to identify the modules with the highest priority. Then these modules are tested to highlight the variables and functions with the highest failure rates. Research in the use of data flow analysis for dependability testing has grown less popular, but by combining the use of data-flow analysis with the results from the fault injection testing, the propagation of the critical errors can be determined and then the important locations and artifacts for implementing error handling mechanisms can be identified. The data-flow analysis helps to keep the error handling code added to the OSS system in only the most critical locations. This should allow for a rapid testing phase that increases the software dependability, without creating too much overhead.

The issue with adding error handling mechanisms this late in the life cycle of the software is that they have a harsh overhead if they are too obtrusive. This thesis proposes using an existing wrapper-based solution, similar to [1], that is a lightweight error handling mechanism that limits this overhead, while still increasing the dependability by a significant amount. The majority of the research for late-stage dependability and reliability techniques is aimed at closed source CotS components. Whereas, the research aimed at systems where the source code is available are intended to be used much earlier in the life cycle. Thus, this methodology creates a means of speeding up the process of locating high priority locations for placing error handling mechanisms to increase the dependability of OSS in order to allow for the benefits

to be used more often in systems that have strict timelines and higher standards for software dependability and reliability. Furthermore, this methodology forms a foundation for a larger methodology that can include increasing desired factors other than dependability, such as security.

Chapter 2

Fault Injection Framework

This chapter details the implementation of the fault injection framework that was used in order to test the methodology on an open source software system.

2.1 Fault Model

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 dependability can be measured.[33] In this methodology, 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 hardware faults. The local model is a transient data value failure, which means that a variable has a corrupted value and could become corrupted again. The global model in this case is the assumption that any variable in the system could be affected by this data failure. This model is the same model used in [1].

2.2 Implementation

The implementation of the fault injection framework was split into three different parts. The first and most crucial part of the framework was the injection functionality that corrupts the value in a variable. The probe functionality is used to output the value of a variable at various locations throughout the testing process. The environment simulator is used to

control the test and manages the output of the tests for comparison to the golden run. Note that all code for the framework was written in C.

2.2.1 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, int *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. This is important when considering several modules in the same system and are discussed in chapter three.

Figure 2.2 shows a simple example that calls the injection function and specifies that the value of 100 be inserted into the variable x and the variable x is then stored in the variable log produced by the framework after each experiment.

The second function, used in the testing in this experiment, injects a random error in the variable. This is done by flipping a random bit in the variable and then returning the 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. Figure 2.3 shows the function to randomly flip a bit in an integer. Note that the file points fp1 and fp2 are defined globally for all functions in the injector file.

```
int injector_i(char *iid, int var, int inject_val, float p);
```

Figure 2.1: Prototype for manual integer injection.

```
int fmul( int x, int y) {
        x = injector_i("mulInject1",x,100,1);
        int result = x * y;
        probe_i("mulProbe1",x);
        return result;}
```

Figure 2.2: Integer injection and probe example.

Figure 2.4 shows the more difficult task of flipping a bit in a floating point variable or pointer. The code is mostly the same between the functions, but the added difficulty of flipping a bit for a variable of floating point or pointer format 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 long double. When the single random bit-flip is complete, the variable is then cast back as the original variable pointer and then dereferenced. This method works with pointers of various types as well, affecting the address to which the variable points.

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.

2.2.2 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

```
//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 + 1;

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

        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;}
        }
```

Figure 2.3: Function for random bit-flip fault in an integer.

```
//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 + 1;

                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;}
                }
```

Figure 2.4: Function for random bit-flip fault in a float.

```
int injector_i(char *iid, int var, int inject_val, float p);
```

Figure 2.5: Prototype for variable probe.

```
//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);}
        }

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

Figure 2.6: Integer variable probe function.

type of variable probe. Figure 2.5 shows the prototype for a probe function and Figure 2.2 already showed an example of the probe function's use in a simple function.

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. Figure 2.6 shows the function for storing the value of the integer.

2.2.3 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.

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 the open source component in the experiment section of this thesis 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 thesis proposes a methodology for accomplishing a task and not a set-in-stone way of completing the tasks. The 'how' of the problem is hypothesized and validated, but the 'what', as in what to use to accomplish the goal, is left up to those who are implementing the methodology.

Chapter 3

Methodology

3.1 Define Critical Path

The first step of the methodology, as shown in Figure 3.2, is to define the critical path for applying error-handling mechanisms. A critical path in this context is a high-priority module and all the functions inside of that module, plus functions in any other module that exchange data with the high-priority module. Determining a critical path is done by finding the highest-priority modules and the functions that deal with that module. Priority is defined as potential impact an error occurring in a module will have on the system.

3.1.1 Instrumentation Stage One

In order to determine the highest-priority module, the target software is instrumented to track the number of calls made to each module and function within that module in one instance of the intended use cases for the system. Finding the frequency that modules will be called with each execution of the target software allows for modules that are not directly related to the proper execution of the system to be eliminated from consideration. This

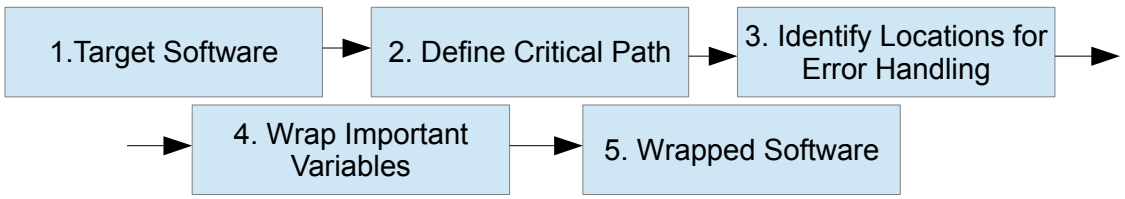

Figure 3.1: Overview of Methodology

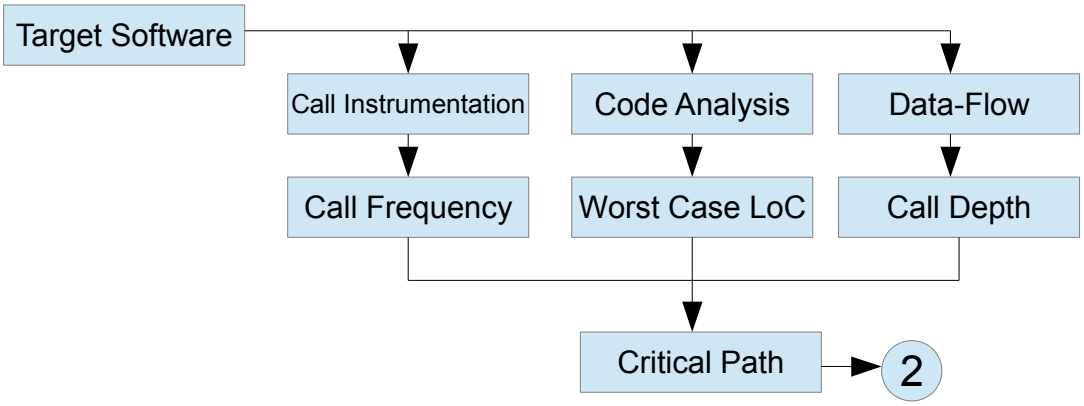

Figure 3.2: Stage one of methodology

narrows down the potential number of modules for testing. Because the focus is on both speeding up the process and accurately identifying important locations for error handling, it is important that no unnecessary tests are completed. Modules with zero or only a fraction of calls compared to other modules will not be considered. The frequency is also found for each function in the remaining modules. This value will be used in a later section to calculate the priority of the function in relation to other functions in the system.

3.1.2 Data Flow Analysis

Once the unrelated modules of the system have been ruled out, it becomes necessary to narrow down the possible high priority locations for error handling further. The next step is to determine the priority of each function in the remaining models. The first value to be determined is the depth of the call and caller graphs for each of the functions. This value will represent the propagation factor of the function in relation to the rest of the functions in the system. For this step, Doxygen was used in order to generate dependency graphs for each module and call and caller graphs for each function. Doxygen also lists any structs, variables, defines, and classes found in each module. Doxygen is most useful when

the target source code has been instrumented with special comment blocks to alert Doxygen that certain information should be included. Even without the extra comment blocks, it is able to generate the necessary graphs.

Doxygen was used through its Linux version command line interface. The setup involves generating a configuration file that has a list of options that the user can either set to 'yes' or 'no' and other options such as giving a specific file path or file names. Doxygen was set up to ignore private internal functions of a module, such as small functions that perform single actions and other such functions. This is because if the function is an internal function, then it is only called by functions within the same module. It has the same propagation to functions in other modules as the function that called it. The Doxygen command is then run again with the configuration file for the target software as input.

This data is important because the call and caller graphs for each function were used to determine its propagation factor in relation to the other potentially high priority functions. The call depth of an individual function was set to the length of the data path from its callers and called functions. The propagation factor was found by equation 3.1, where p is the propagation factor, F_{cd} is the call depth of the function and Max_{cd} is the maximum call depth. The Max_{cd} is highest call depth for all the functions being considered.

$$p = F_{cd}/Max_{cd} \tag{3.1}$$

3.1.3 Determine Priority Functions

When finding the highest priority functions, it was determined that there were three key factors to consider: probability that the single bit-flip fault occurred in memory related to a function, probability that the corrupted code in that function would then be run during the current execution of the program, and the propagation factor. The propagation factor was found as described in the previous section. For the purposes of testing this methodology and proving the concept it is assumed that the probability that the fault occurs in a function to

be equal for all functions. In reality, this is most likely not the case. It should be possible to determine the probability that a single bit-flip fault occurs in the memory space related to a function based on the amount of memory that the function has reserved and obtains dynamically during run-time. This amount of memory would then be divided by the total memory used by the program to determine what percentage of the memory is related to that function. This way is still assuming that the fault has an equal chance to occur in any bit in memory and does not consider which sections of memory would be more prone to having a fault than others.

With the probability that a fault occurs in a function equal for all functions, the remaining factors to be used to set a priority for each function are the propagation factor and the probability that the corrupted code in a function will be run during the same execution of the program in which it occurs. For this thesis, it was determined that the probability that corrupted code in a function would be executed in the instance of the program it was injected would be based on the effective lines of code for that function and the frequency that function was called given the intended use cases. The effective lines of code is found by equation 3.2, where $\mathrm{ELoC_f}$ is effective lines of code. The worst case lines of code for a function, $\mathrm{LoC_{wc}}$, was determined through code analysis as the maximum number of lines of code that could be executed in a single instance of that function and the function call frequency, $\mathrm{F_{cf}}$, was determined through instrumentation.

$$ELoC_f = LoC_{wc} * F_{cf} \tag{3.2}$$

After calculating the effective lines of code for each potentially high-priority function, the probability that corrupted code will be executed in that function at a given point during run-time is found by equation 3.3, where $\mathrm{ELoC_f}$ is the effective lines of code for the function in question and TELoc is the total effective lines of code across all functions.

$$P_e = ELoC_f / TELoC \tag{3.3}$$

At this point for each function, probability of corruption and execution of corrupted code and propagation factor have been determined. These three values are used as shown in equation 3.4 to determine the priority of each function, where P_e is the probability that the corrupted code is executed, P_c is the probability of corruption for the function, and p is the propagation factor. Again, it is assumed that the probability of corruption, P_c is equal for all functions and as such is equal to 1 for all calculations. When considering the potential a fault has to affect a function, it is important to consider both the likelihood that the fault occurs in memory reserved for that function and the likelihood that the corrupted code will even be executed. A fault that occurs in memory causing code corruption in a function that will not be executed again during the remaining run-time of the system, will not have an effect on the outcome of the current run. Priority is determined based on the functions that have the greatest chance to have code corruption that is then executed during run-time and the potential propagation of that fault. The propagation is important because even if the fault does not cause failure immediately in the function in which it was found, further calculations and use of variables that carry the result of the corrupted code could still lead to failure.

$$Priority = P_c * P_e * p \tag{3.4}$$

This priority calculation is then used to determine the priority modules in the system. The priority of the modules is just the sum of the priority calculations for all functions within that module. With the highest priority modules determined, the critical paths can be found. The critical paths are ranked according to the module priorities found for the intended use cases. A critical path is the full call graph for the highest priority function in the high priority module. The first critical path starts from the highest priority function in the system for the intended use cases and the data-flow for the functions it calls and the functions that call it. The second critical path is the same, but for the highest priority function in the second highest priority module, and so on. Testing indicates that determining priority this

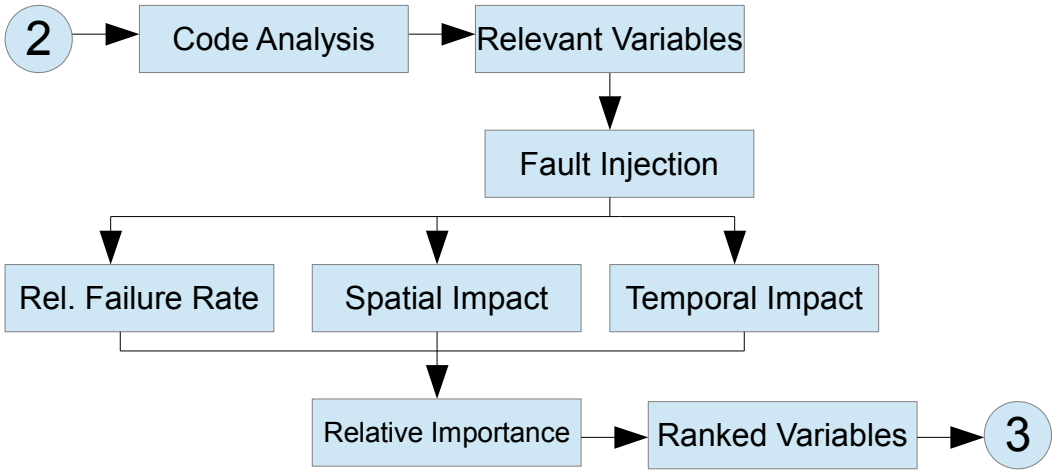

Figure 3.3: Stage two of methodology

way accurately leads to finding the variables that create the highest number of relative failed tests.

3.2 Identify Locations for Error Handling

Now that the critical paths have been determined, it is necessary to identify the priority locations for placing error handling mechanisms. These locations are the instances of reads and writes for the important variables in the critical path. In order to determine these important variables, a variation of the importance metric in [13] is used to calculate the relative importance of all the important variables found in the critical path. This section details the process for ranking the relevant variables, as shown in Figure 3.3.

3.2.1 Determine Important Variables

A technique called dynamic program slicing is used to identify the important variables in the critical path. This is a method of analyzing code that finds all the statements that affect the value of a variable at a given point during a particular execution of the program.

Given the intended use cases determined for testing purposes, the highest priority function and all the statements, and variables within those statements, that have a potential effect on the important variables in the critical path are determined.

First, all the variables in the highest priority function for the critical path are added to the list of relevant variables, because it is unknown which variables in the function will be the most important. But, it is known from the priority calculations from the previous section that the most important variables are in this function. The next step is for each function that is called by the highest priority function in the critical path to be analyzed with dynamic program slicing. In the called functions, the variables that are used in the function call itself are the starting point for the dynamic program slicing. In this called function, any variables that are affected by the variables sent from the original function are added to the list of relevant variables. Then, any variables that affect these variables are added and so on until there are no more variables to be added from the call graph side of the critical path.

For the caller side graph of the critical path, the same method is used to identify relevant variables. The variables sent from the function that calls the highest priority function are added to the list of relevant variables. Then, any variables in program statements that affect these values are added and then the variables that affect those are added and so on. This is done until all variables in the chain of function calls from the high priority function in the critical path to the last function are found. These variables are all added to the list of relevant variables in the critical path and are subjected to fault injection testing.

3.2.2 Instrumentation Stage Two

The second stage of instrumentation of the target source code is for completing the fault injection testing of the relevant variables in the critical path. The goal of the fault injection testing is to determine three key values for each relevant variable in the critical path. These

values are the modified failure rate, spatial impact, and temporal impact. The spatial and temporal impact metrics are defined in [13].

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 in equation 3.5. The spatial impact is then defined as the number of components that are corrupted in r when the variable v is corrupted.

$$\sigma_{v,C} = max\{\sigma^r_{v,C}\}, \forall r \tag{3.5}$$

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. For the purposes of this thesis, only the worst case value of spatial impact is considered for each variable.

Given the same software system, temporal impact as defined by [13] is the impact of a variable v of component C in a run r, denoted in equation 3.6. This value indicates the number of time units where at least one component of the software system remains corrupted in r.

$$\tau_{v,C} = max\{\tau^r_{v,C}\}, \forall r \tag{3.6}$$

After determining the spatial and temporal impact for each relevant variable in the critical path, it is important to determine the relative failure rate of each of the variables. The relative failure rate is the failure rate determined through fault injection when the faults are injected into each function according to the probability of execution found in equation 3.3, given a specific set of use cases for the target software. The reason why this methodology does not use just the local failure rate is that 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. This thesis seeks to modify the importance metric so that its value is relative across multiple modules.

3.2.3 Calculate Relative Importance

The general-form equation for calculating the local importance of a variable is defined in equation 3.7, 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.

$$I_{v,C} = G[K(\sigma_{v,C}), L(\tau_{v,C})] \tag{3.7}$$

The final equation found for the local importance of a variable is as defined in equation 3.8, 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. So, 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.

$$I_{v,C} = [1/(1-f)^n][(\sigma_{v,C}/\sigma_{max} + \tau_{v,C}/\tau_{max})]^m \tag{3.8}$$

The impact of the variables measured in this way is inherently reliant on other components or modules in the system because the spatial and temporal impact are determined based on their effect on these modules. So, in order to make these metrics measure relative importance across multiple modules, σ_{max} and τ_{max} are determined based on the maximum values for all variables in the critical path and not just from within a single function. The

Figure 3.4: Stage three of methodology

other issue of relevance for the importance metric is the failure rate. Finding the failure rate in this way assumes that there is an equal chance of the failure occurring across all functions.

A failure that is introduced into a system, but is never executed has no effect on the outcome of the current run of the program. The failure rate is instead measured with the probability of execution in mind. This means that if the fault is injected at the start of the function to maximize potential damage, the actual likelihood of failure is found only when the fault is injected according to P_{e}. Thus, the relative failure rate is determined by the failure rate of the system when a fault is injected in the function with a probability equal to the P_{e}. These changes yield the relative importance metric as defined in equation 3.9, where σ_{max_r} is the maximum spatial impact across the critical path, τ_{max_r} is the maximum temporal impact across the critical path and f_{r} is the relative failure rate.

$$RI_{v,C} = [1/(1-f_r)^n][(\sigma_{v,C}/\sigma_{max_r} + \tau_{v,C}/\tau_{max_r})]^m \tag{3.9}$$

Once the relative importance, $RI_{v,C}$ has been calculated for all variables in a critical path, they are ranked according this value and then these ranked variables move on to the next stage of the methodology. This process can be repeated for multiple critical paths, where the relative importance is calculated across all paths.

3.3 Wrapper-Based Error Handling

In this last phase of the proposed methodology, wrapper-based error handling mechanisms are put in place around instances of the highest importance variables in the critical

path, as indicated in Figure 3.4. The novelty of this research focuses on identifying important variables for error handling mechanisms and the speed that the identification occurs, not the actual error handling mechanisms. The mechanisms discussed in this section are from the solution in [1], where lightweight and effective error handling mechanisms are tested using a similar testing scenario as is done in this thesis.

The general idea behind the use of the wrapper-based error handling mechanisms with the following algorithms is to utilize lightweight and efficient mechanisms to reduce the failure rate. The lightweight portion of the idea comes from the wrapper-based implementation around the instances of the important variable in a module. This is preferable to an N-programming solution or a wrapper-based on an entire module to secure the correct outcome. The algorithm chosen should be an efficient and well-tested algorithm that is simple to implement and will not introduce a large amount of overhead due to complicated calculations or logic. The user of the methodology must decide upon two threshold values to identify which of the ranked variables should be wrapped.

3.3.1 Write-Wrapper

The first algorithm defined in [1] is the write-wrapper function is called when an important variable is written to. When a variable v is assigned a value f..., where f is some function from the unwrapped module, the ranking of the variable is checked. This is when the selected threshold values come into play. The first threshold, λ_t determines which variables will be triplicated and the second, λ_d determines which will be duplicated. If, for example, the first threshold is the top ten percent out of one hundred variables, then the variables ranked one through ten will have two shadow variables created upon being written to. If the second threshold is fifteen percent, then ranks eleven through fifteen will have a single shadow variable created upon being written to. The remaining variables will not have a wrapper.

Algorithm 1 Write-Wrapper: Writing a variable v

$v := f(...)$
if $(rank(v) \geq \lambda_t)$ **then**
  $create(v')$
  $create(v'')$
  $v, v', v'' := f(...)$
else if $(rank(v) \geq \lambda_d)$ **then**
  $create(v')$
  $v, v' := f(...)$
end if

3.3.2 Read-Wrapper

The second algorithm defined in [1] is the read-wrapper. The read-wrapper is called whenever an important variable is read. So, when a variable y is assigned a value that includes the function$g(v, ...)$ in the unwrapped module, where g is some function of v to be read, the rank of variable v is compared to the threshold values defined earlier. Considering the same example as before, variables ranked one through ten will have a majority algorithm run against the variable v and its two shadow variables, with the majority variable being returned. For variables ranked eleven through fifteen, either the variable v or its single shadow variable will be chosen at random. The algorithms presented in this section are taken from [1] as the focus of the research is in quickly determining important locations for error handling mechanisms based off of the relative importance of variables in the critical path, and not in declaring a new error handling mechanism.

Algorithm 2 Read-Wrapper: Reading a variable v

$y := g(v, ...)$
if $(rank(v) \geq \lambda_t)$ **then**
  $y := g(majority(v, v', v''), ...)$
else if $(rank(v) \geq \lambda_d)$ **then**
  $y := g(random(v, v'), ...)$
end if

3.3.3 Results

This section displays the results of the testing in [1] that indicate that the error handling mechanisms discussed there are sufficient for the purposes of this methodology and do not need to be upgraded at this time. Both tables are taken from [1], where 7Z# are the modules selected for testing from the 7Zip application, FG# are the modules selected for testing from the Flight Gear application and MG# are the modules selected for testing from the Mp3gain application.

Module	Unwrapped Failure Rate	Wrapped Failure Rate
7Z1	0.002407940	0.000017397
7Z2	0.007082023	0.000141946
7Z3	0.000856604	0.000030189
FG1	0.004582688	0.000045475
FG2	0.002481621	0.000002047
FG3	0.001471873	0.000135395
MG1	0.004983750	0.000012083
MG2	0.007888044	0.000013426
MG3	0.002780792	0.000006076

Table 3.1: Results for wrapper-based error mechanism testing [1]

Module	Execution Time	Memory Usage
7Z1	26.048%	07.55%
7Z2	31.470%	18.16%
7Z3	20.359%	00.94%
FG1	30.660%	20.63%
FG2	35.829%	03.32%
FG3	23.529%	02.03%
MG1	25.983%	05.22%
MG2	29.090%	04.93%
MG3	23.174%	00.58%

Table 3.2: Peak increase % in run-time and memory usage [1]

Chapter 4

Experiment

This section details the process of utilizing the proposed methodology in order to find the important error handling locations for an open source software component. The open source software used for this experiment is called Mp3gain. It is a tool that allows the user to normalize volume settings by adjusting the gain of individual mp3 tracks and mp3 tracks across an album. It is a highly modular system and was designed and implemented mostly by a single developer, aside from a few fixes and language translations.

4.1 Design

The set up of the experiment was to test Mp3gain with several potential real world use cases. The use cases selected were the options to scan a track or album for the maximum gain adjustment, normalize volume across all tracks in an album and to undo all known changes to the files by Mp3gain. These were selected as the three test cases to represent the functionality that the hypothetical system looking to incorporate Mp3gain would want to implement. The target source code was instrumented by first adding the header files to each module and then adding the object files from the framework to the path in Mp3gain's makefile. Then, Mp3gain was re-compiled with the instrumentation attached. The first stage of instrumentation was done by having the program output both the module and the specific function within that module to a file. The output was done once for each use case for Mp3gain. The three resulting files were parsed and the frequency of function calls was obtained.

Next, a Doxygen configuration file was created in the Mp3gain directory and setup with the options discussed in chapter 3. The returned results were then used to calculate the call

depth for functions and to aid in determining the critical path once a priority module was determined. The worst case lines of code and dynamic program slicing were done by manual code analysis.

The fault injection testing was completed for each relevant variable found in the critical path. The input to Mp3gain for the three use case tests was an album of 25 tracks. Faults were injected 25 times throughout a single execution of Mp3gain at 25 equal spaced points, meaning that there was one injection per track analysis. Each fault injection test for a variable was completed 100 times, so each use case given 25 input tracks was run 100 times. Two types of runs were completed for each variable. The first run was to obtain the spatial, temporal and failure rate metrics. This was done to obtain the worst case spatial and temporal metrics and to determine the local importance based on local failure rate for validation purposes. The second set of tests for a variable were done using the P_e found earlier to create a number of fault injections relative to the actual expected amount of faults to occur in that function.

Using the relative failure rate and the spatial and temporal impact metrics, the local and relative importance metrics were calculated for each variable. This process was repeated using the second critical path to show that the results of the second set of tests should yield variables with a lesser chance of causing system failure. The number of tests was tracked in order to compare the total number of tests needed to find the important error handling locations using the proposed method to the number of tests needed to complete a cost-benefit analysis of placing error handling in each module in the function. Lastly, the proposed methodology was further validated by comparing the total number of relevant failed test cases found using the proposed methodology versus using local importance.

Note that the wrapper-based error handling mechanisms were not tested in this case. This is because there is no change to the error handling mechanisms suggested for use here compared to those in [1]. The mechanisms proposed there already met the requirements of

being lightweight and efficient, so there was no need to develop new mechanisms. Instead, the focus is on determining important locations for these error handling mechanisms.

4.2 Phase 1

The first phase of testing the methodology was to find the priority modules in the system. Table 4.1 shows the priority of modules and table 4.2 shows the ranking and the values needed to calculate it for the functions in the highest priority modules. Note that in the case of table 4.2, the call frequency of test 1 and test 2 were omitted because there were little to no entries. The majority of the work came from the use case involved the volume normalization across the album, not the max gain analysis or undoing the changes. This was due to the tag that Mp3gain updates on each file whenever it makes a change.

Table 4.1: Priority of Modules

Module	T1 Freq	T2 Freq	T3 Freq	Sum Priority	Rank	# of variables
mp3gain	1	462862	483097	3841.5	7	168
apetag	109	151	121	521.7	8	58
gain_analysis	0	0	46572341	202183.975	6	53
id3tag	0	0	0	0	9	87
replaygaindll	0	0	0	0	9	19
rg_error	0	0	0	0	9	6
common	0	0	4350007	9207297.75	2	24
decode_i386	0	0	1448424	55257375.6	1	18
dct64_i386	0	0	2896848	2027793.6	5	11
interface	4	0	241416	2908720.125	3	37
layer1	0	0	0	0	9	24
layer2	0	0	0	0	9	54
layer3	2	0	123710987	2684186.725	4	65
tabinit	0	0	0	0	9	16

The knowledge obtained from these tables is where to start the analysis for the critical path. In this thesis, testing was done for the first and second critical path so that the results could be analyzed to ensure that the first critical path did indeed lead to more failed test cases than the second. The last step of this phase was determining the critical path. Figure 4.1 through Figure 4.3 indicate the critical path for the highest priority module in the path

Table 4.2: High Priority Module - Functions

Module	Function	T3 Freq	Call Depth	Rank	ELOC	P_e
mp3gain	main	1	40	15	3693	0.0000075323
common	head_check	20117	4	11	905265	0.0018463889
common	decode_header	20117	8	7	1830647	0.0037338086
common	print_header	0	1	31	0	0.0000000000
common	print_header_compact	0	1	31	0	0.0000000000
common	getbits	4289656	3	2	102951744	0.2099815579
common	getbits_fast	2209014	1	6	39762252	0.0810995452
common	set_pointer	20117	5	13	281638	0.0005744321
decode_i386	synth_1to1_mono	0	7	31	0	0.0000000000
decode_i386	synth_1to1	1448424	7	1	315756432	0.6440204404
dct64_i386	dct64	2896848	7	4	11587392	0.0236337776
interface	InitMP3	2	5	24	128	0.0000002611
interface	ExitMP3	2	3	27	44	0.0000000897
interface	remove_buf	20117	4	14	281638	0.0005744321
interface	copy_mp	60351	4	10	1267371	0.0025849444
interface	GetVbrTag	2	4	22	190	0.0000003875
interface	check_vbr_header	2	4	25	72	0.0000001469
interface	sync_buffer	20117	4	9	1448424	0.0029542222
interface	decodeMP3	20119	21	3	4969393	0.0101356310
layer3	init_layer3	2	4	19	760	0.0000015501
layer3	do_layer3_sideinfo	20117	3	12	865031	0.0017643271
layer3	do_layer3	40234	8	5	5672994	0.0115707036
layer3	III_get_side_info_2	20117	2	13	1750179	0.003569685
layer3	III_dequantize_samples	80468	2	5	29692692	0.0605615552
		MAX CD	40	SUM ELOC =	490289457	—

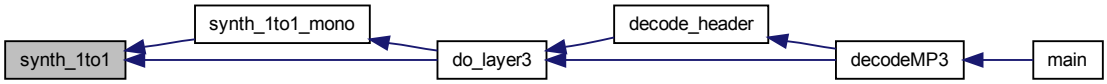

Figure 4.1: Caller side for first critical path

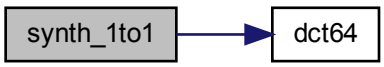

Figure 4.2: Called side for first critical path

as denoted by a grey box. Figure 4.4 indicates the caller graph for the function getbitsfast, which was included to validate that there were no important variables in that path and only in the path defined by the highest priority function, getbits.

4.3 Phase 2

The second phase of the experiment tested the variables acquired through dynamic program slicing for spatial and temporal impact, failure rate, and relevant failure rate. Note that in the data presented here, temporal impact was limited to the value of either 1 or 25.

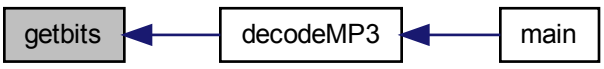

Figure 4.3: Call graph for second critical path

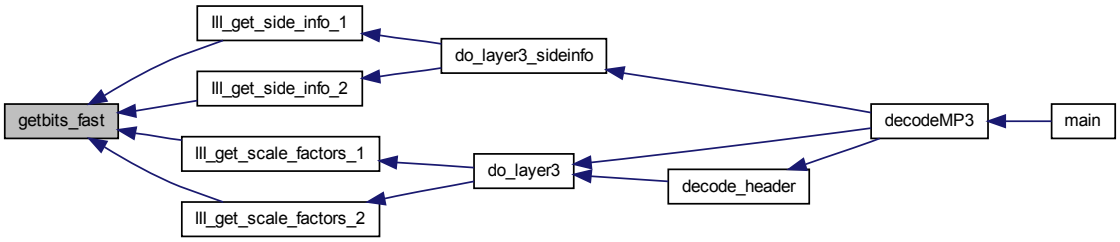

Figure 4.4: Call graph for alternate second critical path

The failures either caused a failure in the single track into which they were injected into or caused the entire album normalization process to fail, such as with a segmentation fault.

Both of the critical paths were subjected to the fault injection testing and the results of the top 15% of the variables are included in Table 4.1 and 4.2. Critical path one has 47 variables total and critical path two, including the additional branch from getbitsfast that would not usually be considered, has 59 variables. The additional variables from the getbitsfast path are included to show that none qualify as an important variable. The only variables from critical path two that are near the 15% threshold of rank 16 is the rank 17 bsbufold and rank 7 wordpointer variable. The other variables are either out of range or already located in critical path one.

The variables analyzed in critical path one are only those which have the potential to cause an error based on their connection to the highest priority module in the system. It is understandable that there should be some variables located in critical path two or even three that have importance rankings not too far out from the thresholds. But, it is clear that the majority of high importance variables found through the fault injection are located in the first critical path. These highest ranking variables are the key to setting error handling mechanisms. The important locations for setting error handling mechanisms are the code statements that either write to or read from one of the ranking variables within the defined thresholds. Also, by expanding the range of critical path two and incorporating the variables

from other functions in the dolayer3 module more variables were found for fault injection testing. None of the variables were within the importance ranking threshold. There was one rank 30 variable and the rest were between rank 40 and 80. Selecting the variables from the program slice of the highest priority function, getbits, was more effective at finding the important variables than the variables from the program slice of the second highest priority function, getbitsfast.

Function	Variable	$\sigma_{v,C}$	$\tau_{v,C}$	Failure Rate	Relative Failure Rate	Relative Importance
synth_1to1	bandPtr	3	25	0.0138666667	0.0088	1.7816802003
synth_1to1	pnt	3	25	0.0088	0.0056	1.769765878
do_layer3	pcm_point	3	25	0.0284	0.0002666667	1.7511506951
synth_1to1	buf	2	25	0.0321333333	0.0206666667	1.5640653421
synth_1to1	maxAmpOnly	2	25	0.0081333333	0.0052	1.5158384332
main	curframe	2	25	0.0188	0	1.5
synth_1to1	window	1	25	0.0734666667	0.0472	1.3772426984

Table 4.3: Fault injection results for variables for critical path one

Function	Variable	$\sigma_{v,C}$	$\tau_{v,C}$	Failure Rate	Relative Failure Rate	Relative Importance
main	curframe	2	25	0.0188	0	1.5
common	wordpointer	1	25	0.188	0.0548	1.3992370494
common	bsbufold	1	25	0.1876	0	1.2502694522
main	wrdpntr	1	25	0.0033333333	0	1.25
main	nprocsamp	4	1	0.0017333333	0	1.04
common	rval	2	1	0.2857333333	0.0830666667	0.6424178797
common	number_of_bits	2	1	0.1666666667	0.0484	0.5964698746
layer3	h− >linbits	2	1	0.0217333333	0.0012	0.5414243122
main	bytesinframe	2	1	0.0065333333	0	0.54

Table 4.4: Fault injection results for variables for critical path two

Chapter 5

Results

To validate the results obtained in chapter 4, testing was done to determine the number of relative failed tests using the proposed methodology versus using just local importance for cost-benefit analysis. In this context, a failed test was determined using the standard fault injection method and a relative failed test was determined using the fault injection based on the probability of execution. Table 5.1 and Table 5.2 show the number of failed tests and relative failed tests found by the proposed method and by just using local importance, respectively. Table 5.3 shows the percent difference between the proposed method and a cost-benefit analysis of placing error handling mechanisms using only local importance.

Table 5.1: Number of failed tests found using proposed methodology

Critical Path #	Failed Tests	Relative Failed Tests
CP1	2135	1145
CP2	6601	1406
CP1/2	5398	1724

Table 5.2: Number of failed tests found using just the local importance

Critical Path #	Failed Tests	Relative Failed Tests
CP1	3647	110
CP2	6551	1110
CP1/2	9100	1155

Just using local importance as a metric to assigning locations for wrapper-functions is unreliable and is influenced primarily by failure rate and not by other factors, such as spatial and temporal impact across multiple modules, or the likelihood that the code corruption

Table 5.3: % difference between the number of failed tests found

Critical Path #	%diff FR	%diff MFR
CP1	-70.8196721311	90.3930131004
CP2	0.7574609908	21.0526315789
CP1/2	-68.5809559096	33.0046403712

Table 5.4: Proposed Methodology vs. Local Cost-Benefit Analysis

MP3gain	# of tests PM	# of tests LCBA	% diff
CP1	47	640	92.65625
CP1/2	97	640	84.84375

in that function is meaningful. The proposed methodology does not appear to generate satisfactory results. The relative importance metric accounts for the highest possible spatial impact and temporal impact, as well as the likelihood that the corrupted code will be run in the function where the testing is taking place. By injecting faults with a probability equal to the chance that the function will still be run at the time of injection, the number of failures found by the proposed method ranges from 21% to 90% greater than the number of failures found just by local importance. This is meaningful because calculating the failure rate when the fault is known to be placed into a particular location for testing purposes, the resulting failure rate loses its relevance to the system as a whole. One must also consider the chance that the fault will even occur in the memory space belonging to that function and the chance that when the fault occurs, the resulting code corruption will even be executed.

Also, Table 5.4 clearly shows a significantly reduced number of tests required to locate these important error handling locations. Given these results, the methodology would appear to accurately locate variables with a high failure rate relative to the intended use cases for the system, as well as drastically reduce the number of tests required to identify importance error handling locations in the target software. There is still room for improvement in the design

and implementation of the methodology, discussed in chapter 6, but the results indicate a successful completion of the research goal.

Chapter 6

Conclusion and Future Work

The primary benefit this research provides is to create a methodology for quickly finding important locations for placing lightweight error handling mechanisms. These mechanisms will then increase the dependability of the OSS components they are placed into, allowing systems that have higher standards for dependability and stricter timelines to consider using OSS components. This is an important benefit that will serve as a step towards increasing the use of OSS in more real world applications and mission critical systems. OSS has many clear benefits, the most applicable here being the savings in both time and money that are not being fully utilized in the current market.

The proposed methodology also provides data from a hybrid testing process that combines a fault injection environment with data-flow analysis to identify critical modules and artifacts in the software by mapping out their propagation throughout the software to establish a critical path. This path can then be used to illustrate the most critical variables of the software and how they communicate with each other. These critical variables are determined through a measurement of their importance determined primarily by relative failure rate and the flow of data between and within modules. This serves to show the benefits of combining the two testing processes for greater effect and also to further verify the concept of determining the importance of artifacts in the software based on their failure rates and propagation.

The shortcomings of current methodologies are that they are intended for use in earlier stages of the design process and do not allow for bias in the modular structure of the system. That is, all modules and signals are treated with the same level of importance in the greater system view. The proposed methodology attempts to incorporate an understanding of system

structure, dependability properties and insight into the operation of the system as a whole to help determine placement of the error handling mechanisms. For instance, in the case that there is a hub module that the majority of modules receive input from or certain signals that have a greater effect on the overall system, the proposed methodology will recognize these through code analysis and data-flow analysis. Instead of treating each module as an isolated unit, the proposed methodology also considers the flow between modules before instrumenting the additional error handling mechanisms into the source code.

Lastly, this research is directed at the level of dependability of the OSS systems being tested, but it would be possible to focus the research instead on the level of security or other possible factors. The methodology proposed in this thesis will form a foundation for a more refined methodology that could be molded to fit the needs of the system to be integrated into, or possibly a combination of dependability and security. There is still refinement to be done with the current goal of quickly and accurately defining high importance locations for placing error handling mechanisms. Considerations should be made as to the probability that a fault occurs in memory related to a function and how that would change the priority of the modules. Further automation of the designed fault injection framework and instrumentation process should be pursued in the case that the methodology is to see any widespread use on large components or systems.

Bibliography

[1] Leeke, M., Jhumka, A. An Automated Wrapper-based Approach to the Design of Dependable Software. in DEPEND 2011:The Fourth International Conference on Dependability.(Cte d'Azur, France 2011) 43-50.

[2] Arora, A., Kulkarni, S. Detectors and Correctors: A Theory of Fault-Tolerance Components. in 18th International Conference on Distributed Computing Systems.(May 1998) 436-443.

[3] Hiller, M., Jhumka, A., Suri, N. PROPANE: An Environment for Examining the Propagation of Errors in Software. in ACM SIGSOFT International Symposium on Software Testing and Analysis.(New York, USA 2002) 81-85.

[4] Hsueh, M., Tsai, T., Iyer, R. Fault Injection Techniques and Tools. ed. Computer, 30 (4). 75-82. April 1997.

[5] Sosnowski, J., Gawkowski P., Zygulski P., Tymoczko A. Enchancing Fault Injection Testbench. in International Conference on Dependability of Computer Systems.(May 2006) 76-83.

[6] Moraes, R., Duraes, J., Barbosa, R., Martins, E., Madeira, H. Experimental Risk Assessment and Comparison Using Software Fault Injection. in 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. (June 2007) 512-521.

[7] Natella, R., Cotroneo, D., Duraes, J., Madeira, H. On Fault Representativeness of Software Fault Injection. in IEEE Transactions on Software Engineering. 37 (6). 1-18. 2011.

[8] Leeke, M., Jhumka, A. The Early Identification of Detector Locations in Dependable Software. in 22nd IEEE International Symposium on Software Reliability Engineering. (Hiroshima, Japan 2011) 40-49.

[9] Leeke, M., Arif, S., Jhumka A., Annad, S. A Methodology for the Generation of Efficient Error Detection Mechanisms. in 41st IEEE/IFIP International Conference on Dependable Systems and Networks. (Hong Kong, China 2011) 25-36.

[10] Cinque, M., Cotroneo, D., Natella, R., Pecchia, A. Assessing and Improving the Effectiveness of Logs for the Analysis of Software Faults. in IEEE/IFIP International Conference on Dependable Systems and Networks. (Chicago, USA 2010) 457-466.

[11] Clark, J., Pradhan, D. Fault Injection: A Method for Validating Computer-System Dependability. ed. Computer, 28 (6). 47-55. June 1995.

[12] Madeira, H., Costa, D., Vieira, M. On the Emulation of Software Faults by Software Fault Injection. in International Conference on Dependable Systems and Networks. (New York, USA 2000). 417-426.

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

[14] Near, J., Milicevic, A., Kang, E., Jackson, D. A Lightweight Code Analysis and Its Role in Evaluation of a Dependability Case. in 33rd International Conference on Software Engineering.(Honolulu, USA 2011). 31-40.

[15] Lisboa, C., Grando, C., Moreira, A., Carro, L. Using Software Invariants for Dynamic Detection of Transient Errors. in 10th Latin American Test Workshop. (Rio de Janeiro, Brazil 2009). 1-6.

[16] Hamill, M., Goseva-Popstojanova, K. Common Trends in Software Fault and Failure Data. ed. IEEE Transactions on Software Engineering, 34 (4). 484-496. August 2009.

[17] Song, N., Qin, J., Pan, X., Deng, Y. Fault Injection Methodology and Tools. in International Conference on Electronics and Optoelectronics (ICEOE). (Dalian, China 2011). V1-47 V1-50.

[18] Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J., Laprie, J., Martins, E., Powell, D. Fault Injection for Dependability Validation: A Methodology and Some Applications. ed. IEEE Transactions on Software Engineering, 16 (2). 166-182. February 1990.

[19] Barton, J., Czeck, E., Segall, Z., Siewiorek, D. Fault Injection Experiments Using FIAT. ed. IEEE Transactions on Computers, 39 (4). 575-582. April 1990.

[20] Benso, A., Carlo, S. The Art of Fault Injection. ed. Journal of Control Engineering and Applied Informatics, 13 (4). 9-18, August 2011.

[21] Jiang, M., Munawar, M., Reidemeister, T., Ward, P. Automatic Fault Detection and Diagnosis in Complex Software Systems by Information-Theoretic Monitoring. in IEEE/IFIP International Conference on Dependable Systems and Networks, (Estoril, Portugal 2009). 285-294.

[22] Natella, R. Achieving Representative Faultloads in Software Fault Injection. Universita Degli Studi Di Napoli Federico Il, November 2011.

[23] Leeke, M., Jhumka, A. Evaluating the Use of Reference Run Models in Fault Injection Analysis. in IEEE Pacific Rim International Symposium on Dependable Computing, (Shanghai, China 2009). 121-124.

[24] Voas, J. Certifying Off-the-Shelf Software Components. ed. Computer, 31 (6). 53-59, June 1998.

[25] Kropp, N., Koopman, P., Siewiorek, D. Automated Robustness Testing of Off-the-Shelf Software Components. in 28th Annual International Symposium on Fault-Tolerant Computing, (Munich, Germany 1998). 230-239.

[26] Voas, J., Ghosh, Anup. Inoculating Software for Survivability. ed. Communications of the ACM, 42 (7). July 1999.

[27] Khanjani, A., Sulaiman, R. The Aspects of Choosing Open Source versus Closed Source. ed. IEEE Symposium on Computers and Informatics, (Kuala Lumpur, Malaysia 2011). 646-649.

[28] Nagy, D., Yassin, A., Bhattacherjee, A. Organizational Adoption of Open Source Software: Barriers and Remedies. ed. Communications of the ACM, 53 (3). 148-151. March 2010.

[29] Xiong, L., Tan, Q. Data Flow Error Recovery with Checkpointing and Instruction-level Fault Tolerance. in 12th International Conference on Parallel and Distributed Computing, Applications and Technologies, (Gwangju, Korea 2011). 79-85.

[30] Taylor, R., Osterweil, L. Anomaly Detection in Concurrent Software by Static Data Flow Analysis. ed. IEEE Transactions on Software Engineering, 6 (3). 265-278. May 1980.

[31] Bergeretti, J., Carre, B. Information-Flow and Data-Flow Analysis of while-Programs. ed. ACM Transactions on Programming Languages and Systems, 7 (1). 37-61. January 1985.

[32] Bovenzi, A., Cotroneo, D., Pietrantuono, R., Carrozza, G. Error Detection Framework for Complex Software Systems. in 13th European Workshop on Dependable Computing, (Pisa, Italy 2011). 61-66.

[33] H. Volzer, Verifying fault tolerance of distributed algorithms formally - an example, in Proceedings of the 1st International Conference on the Application of Concurrency to System Design, March 1998, 187-197.

[34] E. Capra, C. Francalanci, F. Merlo, C. Rossi-Lamastra. Firms' involvement in Open Source projects: A trade-off between software structural quality and popularity. Journal of Systems and Software 84 (1), January 2011, 144-161.