View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Warwick Research Archives Portal Repository

WARWICK
THE UNIVERSITY OF WARWICK

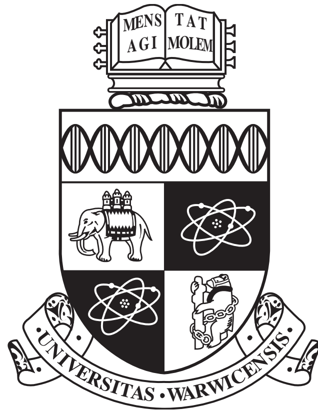
A Thesis Submitted for the Degree of PhD at the University of Warwick

Permanent WRAP URL:
http://wrap.warwick.ac.uk/92013

Copyright and reuse:
This thesis is made available online and is protected by original copyright.
Please scroll down to view the document itself.
Please refer to the repository record for this item for information to help you to cite it.
Our policy information is available from the repository home page.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk

warwick.ac.uk/lib-publications



$\mathbf{L_nC_m}$ Fault Model: Complexity and Validation

by

Fatimah Adamu-Fika

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

September 2016

Abstract

Computer systems are ubiquitous in most aspects of our daily lives, as such the reliance of end users upon their correct and timely functioning is on the rise. With technology advancement, the functionality of these systems is increasingly being defined in software. On the other hand, feature sizes have drastically decreased, while feature density has increased. These hardware trends will keep happening as technology continues to advance. Consequently, power supply voltage is ever-decreasing and clock frequency and temperature hotspots are increasing. This steady reduction of integration scales is increasing the sensitivity of computer systems to different kinds of hardware faults. In particular, the likelihood of a single high-energy ion to cause double bit upsets (DBUs, due to its energy) or multiple bit upsets (MBUs, due to the incident angle) instead of single bits upsets (SBUs) is increasing. Furthermore, the likelihood of perturbations occurring in the logic circuits is also increasing. Owing to these hardware trends it has been projected that computer systems will expose such hardware faults to the software-level and accordingly the software is expected to tolerate such perturbations to maintain correct operations, i.e., the software needs to be dependable. Thus, defining and understanding the potential impact of such faults is required to propose the right mechanisms to tolerate their occurrence. To ascertain that software is dependable, it is important to validate the software system. This is achieved through the emulation of the type of faults that are likely to occur in the field during execution of the system, and through studying the effects of these faults on the system. Often, this validation process is achieved through a technique called fault injection, that artificially perturbs the execution of the system through the emulation of hardware faults.

Traditionally, the single bit-flip (SBF) model is used for emulating single event upsets (SEUs) and single event transients (SETs) in dependability validation. The model assumes that only an SEU or SET occurs during a single execution of the system. However, with MBUs becoming more prominent, the accuracy of the SBF model is limited. Hence, the need for including MBUs in software system dependability validation. MBUs may occur as multiple bit errors (MBEs) is a single location (memory word or register) or as single bits errors (SBEs) in several locations. Likewise, they may occur as MBEs in several locations.

In the context of software-implemented fault injection (SWIFI), the injection of MBUs in all variables is infeasible due to the exponential size of the fault space, thereby making it necessary to carefully select those fault injection points that maximises the probability of causing a failure. A fault space, is the set of all possible fault under a given fault model. Consequently, research have started looking at a more tractable model, double bit upsets (DBU) in the form of *double bit-flips* within a single location, L_1C_2. However, with evidence of the possibility of corruption occurring chip wide, the applicability and accuracy of L_1C_2 is restricted. Following, this research focuses on MBUs occurring across multiple locations whilst seeking to address the exponential fault space problem associated with multiple fault injections.

In general, the thesis analyses the complexity of selecting efficient fault-injection locations[1] for injecting multiple MBUs. In particular, it formalises the problem of multiple bit-flip injections and found that the problem is NP-complete. There are various ways of addressing this complexity: (i) look for specific cases, (ii) look for heuristic and/or (iii) weaken the problem specification.

Next, the thesis presents one approach for each of the aforementioned means of addressing complexity:

- for the specific cases approach, the thesis presents a novel DBU fault

[1] injection points that would uncover vulnerabilities and/or cause system failure.

model, that manifest as two single bit-flips across two locations. In particular, the research examines the relevance of the L_2C_1 fault model for system validation. It is found that the L_2C_1 fault model induces failure profile that is different from profiles induced by existing fault models.

- for the heuristic approach, the thesis uses an approach towards dependency aware fault injection strategies to extend the L_2C_1 fault model and the existing L_1C_2 fault model into L_nC_m (multiple location, multiple corruption) fault model, where n is the number of locations to target and m the maximum number of corruptions to inject in a given location. It proposes two heuristics to achieve this: first, select the set of potential locations and then select the subset of variables within these locations, and it examines the applicability of the proposed framework.

- for the weakening of the problem specification approach, the thesis further refines the fault space and proposes a data mining approach to reduce the cost of multiple fault injections campaigns (in terms of number of multiple fault injections experiments performed). It presents an approach to refine the multiple fault injection points by identifying a subset of these points, whereby injection into this subset alone would be as efficient as injection into the entire set.

These contributions are instrumental to advance multiple fault injections and make it an effective and practical approach for software system validation.

To my Mamah and Baba,

For their endless love, support and encouragement.

Also to the memory of

my dear Iya, Khadijah Tasalla Fika,

my favourite uncles, Emir Abali Ibn Muhamad

& Mallam Jibril Amfani,

my darling aunt, Zainabu “Ya Abu” Amfani,

and

my dear cousins, Bilkisu Fika,

Adamu “Taju” Bala Fika & Haruna Abali.

Acknowledgements

First and foremost, I give my utmost gratitude to God for letting me reach this point of my graduate education.

I am forever indebted to Islamic Development Bank for granting me a scholarship to undertake my graduate studies, and also to Yobe State government for their financial aid. Without these grants it wouldn't have been possible to be writing this thesis now.

Throughout my graduate education Dr. Arshad Jhumka has been an excellent supervisor, mentor and most of all friend. He saw what I did not see in myself and pushed me hard to achieve what he knew I could. I doubt very much I would gotten to this stage with out his guidance and support. I would also like to thank my former supervisor, Sarab Singh, for accepting me as a student, and for being a friend and mentor.

I am especially thankful to my mentors Dr. Ardo Bamanga, Mr. Musa Maina Mshelia and Mr. Lekan Muyiwa Ogedengbe, for their unwavering support and advise. I thank you all from the bottom of my heart. My sincerest gratitude to keen sounding boards Hadiza Fika, Hadi Fika and Hassana Fika-Mohammed, for always having time for (and never getting irritated by) my constant whining and nagging.

I would also like to express my gratitude to my colleagues who turned friends, who motivated me through out my graduate journey - especially, Saima Arif, Nentawe Gurumdimma, Adekunle Shonola, Daniel Onah and Daniel Nwaigwe. I am also thankful to the entire staff of the Computer Science Department,

especially the members of the Systems and Software group.

To my friends, old and new - most especially, Maryam Uwani Abdullahi, Safiya & Sharif Abdullahi, Nana & Jonathan Lyamgohn, Asmau “Aims” Smaila, Sani Sidi, Alheri Loma, Fatima Goje, Ibrahim Muazzam, Abdul Isa Waziri, Jamilla Bello and my LIS ’95 and Gwags tribes - I say a big thank you! You all helped my sanity remain sane through this arduous journey.

Not least of all, I owe so much to my whole family and family friends for their undying support, their unwavering belief that I can achieve so much. Unfortunately, I cannot thank everyone by name because it would take a lifetime but, I just want you all to know that you count so much. Had it not been for all your prayers and benedictions; were it not for your sincere love and help, I would never have completed this thesis. So thank you all.

Declarations

This thesis include and extends materials from the following works:

[2] F. Adamu-Fika and A. Jhumka. An investigation of the impact of double single bit-flip errors on program executions. In P. Lorenz and F. P. Dini, editors, *DEPEND 2015, The Eight International Conference on Dependability*, pages 15 – 22, Venice, Italy, August 2015. IARIA. ISBN 978-1-61208-429-9. URL http://www.thinkmind.org/index.php?view=article&articleid=depend_2015_1_40_50038

[1] F. Adamu-Fika and A. Jhumka. *Algorithms and Architectures for Parallel Processing: 15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18-20, 2015, Proceedings, Part IV*, chapter An Investigation of the Impact of Double Bit-Flip Error Variants on Program Execution, pages 799–813. Springer International Publishing, Cham, 2015. ISBN 978-3-319-27140-8. doi: 10.1007/978-3-319-27140-8_55. URL http://dx.doi.org/10.1007/978-3-319-27140-8_55

Sponsorship and Grants

The research presented in this thesis was made possible by the support of the following benefactors and sources:

- Islamic Development Bank:
  Merit Scholarship Programme for High Technology (MSP)
  (2011–2014)
- Yobe State, Nigeria:
  PhD Scholarship Grant
  (2014–2015)

Abbreviations

ARFF	Attribute-Relation File Format
ALU	Arithmetic Logic Unit
AUC	Area Under ROC Curve
CFG	Control Flow Graph
COTS	Commercial-Off-The-Shelf
CPU	Central Processing Unit
DBU	Double Bit Upsets
DF	Dominance Frontier
DRAM	Dynamic Random Access Memory
EXP	Exponential Time
FI	Fault Injection
FIT	Failures-In-Time
FN	False Negative
FNR	False Negative Rate
FP	False Positive
FPR	False Positive Rate
IC	Intergrated Circuit
ILS	Injection Location Selection
ISA	Instruction Set Architecture
$\mathbf{L_nC_m}$	Multiple-Locations Multiple Corruptions
LLFI	Low Level FI
LLVM	Low Level Virtual Machine
MATLAB	MATrix LABoratory
MBF	Multiple Bit-Flips

MBU	Multiple Bit Upsets
MDS	Minimum Dominating Set
MDT	Mean Down Time
MEU	Multiple Event Upsets
MET	Multiple Event Transients
MRI	Magnetic Resonance Imaging
MSB	Multiple Single Bit-Flips
MTBF	Mean Time Between Failures
MTVS	Minimum TVS
MVC	Minimum Vertex Cover
NP	Non-deterministic Polynomial Time
P	Polynomial Time
PGM	Portable Gray Map
SER	Soft-Error Rate
SEU	Single Event Upset
SBF	Single Bit-Flip
SBU	Single Bit Upset
SD	Standard Deviation
SDC	Silent Data Corruption
SEU	Single Event Upset
SEU	Single Event Transient
SFI	Software FI
SRAM	Static Random Access Memory
SWIFI	Software-Implemented FI
SIHFT	Software-Implemented Hardware Fault Tolerance
ROC	Receiver Operating Characteristic
TN	True Negative
TNR	True Negative Rate
TP	True Positive

TPR True Positive Rate

TVS Target Variable Set

Contents

Abstract i

Dedication iv

Acknowledgements v

Declarations vii

Sponsorship and Grants viii

Abbreviations ix

List of Figures xxi

List of Tables xxv

List of Algorithms xxvi

1 Introduction 1

1.1 Motivations 3

1.2 Thesis Contributions 4

1.3 Thesis Structure 6

2 (Software) Dependability Concepts and Terminology 8

2.1 The Fundamentals of Dependability 8

2.1.1 Dependability Attributes 9

2.1.2 Dependability Threats 12

2.1.3 Type of Faults 13

2.1.4 Dependability Means 15

2.2 Fault Tolerance Validation 17

2.2.1 Formal Method 18

2.2.2 Fault Injection 19

2.2.3 Dependability Analysis 22

3 System and Faults Models and Target Systems 23

3.1 System Model 23

3.1.1 Extended-CFG for a Program 25

3.2 Fault Model 26

3.2.1 Single Fault 27

3.2.2 Multiple Faults 27

3.3 Target Systems 29

3.3.1 Flight Control 29

3.3.2 SUSAN (Smallest Univalue Segment Assimilating Nucleus) 30

3.3.3 MiBench Suite 30

3.4 Fault Injection Analysis 34

3.4.1 LLVM 34

3.4.2 LLVM Fault Injection (LLFI) Tool 34

3.4.3 Failure Scheme 37

4 Problem Statements 38

4.1 Selecting Potential Injection Blocks Locations 39

4.2 Identifying Candidate Variables to Target 39

4.2.1 Error Propagation Masking 40

4.2.2 Error Propagation Amplification 42

4.3 Selecting Choice Bit-Positions 44

4.4 Roadmap of Thesis Statement 45

5 Towards Selecting Locations for Multiple Soft-Errors Injection 48

5.1 Basic concepts of Computational Complexity Theory 50

5.1.1 Reducibility, NP-hardness and NP-completeness 51

5.2 Selecting Locations for Mulitple Fault Injections 53

5.3 Injection Location Selection (ILS) 54

5.3.1 Complexity Analysis of ILS 55

5.4 Target Variable Selection (TVS) 59

5.4.1 Complexity Analysis of TVS 59

5.5 Summary and Conclusions 62

6 Double Single Bit-Flips (L_1C_2) Fault Model 64

6.1 Evaluation of Fault Models and Failure Modes 66

6.2 Case Studies 67

6.2.1 System Instrumentation 68

6.2.2 Experimental Procedure 69

6.3 Impact of Fault Models 72

6.3.1 L_2C_1 vs. L_1C_2 vs L_1C_1 76

6.4 Impact of Injection Location 81

6.4.1 Block Location 82

6.4.2 Register Instruction Type 86

6.4.3 Register Data Type 87

6.5 Correlations 89

6.5.1 Testing Monotonic Relationships 93

6.5.2 Testing Linear Relationships 95

6.6 Implication and Limitation 96

6.7 Summary and Conclusions 96

7 Towards Efficient Multiple Soft-Errors Injection 98

7.1 Selecting Locations for Mulitple Fault Injections 100

7.2 Injection Location Selection (ILS) 101

7.2.1 Heuristic for ILS 102

7.3 Target Variable Selection (TVS) 105

7.3.1 Heuristic for TVS 106

7.4 Case Studies 107

7.5 Experiment Setup 109

7.5.1 Application of the Proposed Framewwork 110

7.5.2 Experiment Procedure 117

7.6 Evaluation of the Case Studies 118

7.6.1 Variable Selection Method Effects 120

7.6.2 Fault Model Effects 131

7.7 Implication and Limitation 134

7.8 Summary and Conclusions 136

8 Learning Bits Patterns 138

8.1 Data Mining in Software Dependability 139

8.2 Data Mining Concepts 140

8.2.1 Fundamentals of Data Mining 140

8.3 Assessment Metrics for Model Quality 145

8.4 Addressing Class Imbalance 150

8.5 Generating Fault Injection Points 153

8.5.1 Stage 1: Data Preparation 154

8.5.2 Stage 2: Model Fitting 156

8.5.3 Stage 3: Model Optimisation 157

8.6 Case Studies 157

8.6.1 Stage 1: Data Preparation 158

8.6.2 Stage 2: Model Fitting 159

8.6.3 Stage 3: Optimising Model 167

8.6.4 Bit-position and injection efficiency 171

8.7 Implication and Limitation 176

8.8 Summary and Conclusions 177

9 Conclusions **179**

9.1 Research Contribution Summary 179

9.1.1 Complexity Analysis and Formalisation of ILS and TVS Problem 180

9.1.2 Double Single Bit-Flips Fault Model 180

9.1.3 Heuristics for the Injection Locations and Target Variables Selection 180

9.1.4 Efficient Bit Locations 181

9.2 Applications 181

9.3 Future Work 182

Bibliography 184

List of Figures

2.1 Dependability tree. [7] 9

2.2 An overview of fault and error terminology focused on the transient hardware fault. 14

2.3 An overview of the relationshipship of an 8-bit MBU and a 3-bit data word with an 8-bit interleave. 15

2.4 Block diagram of propagation of soft-error impacting software. . . 15

2.5 Overview of fault tolerance coverage [7]. 18

2.6 An overview of a basic fault injection environment. 20

2.7 An overview of fault tolerance validation. 20

2.8 An overview of dependability analysis. 22

3.1 Example of basic blocks for a dummy program and its corresponding CFG. 24

3.2 LLFI workflow [156]. 35

4.1 An overview of the thesis contributions. 45

6.1 An overview of double faults. 65

6.2 Error sensitivity distribution of the various fault models for each programs. 73

6.2 Error sensitivity distribution of the various fault models for each program. 74

6.2 Error sensitivity distribution of the various fault models for each program. 75

6.3 Error sensitivity distribution of instruction type for the fault models over all target programs. 86

6.3 Error sensitivity distribution of instruction type for the fault models over all target programs. 87

6.3 Error sensitivity distribution of instruction type for the fault models over all target programs. 88

6.4 Error sensitivity distribution of data type for the fault models over all target programs. 89

6.4 Error sensitivity distribution of data type for the fault models over all target programs. 90

6.4 Error sensitivity distribution of data type for the fault models over all target programs. 91

7.1 Example of a dominator tree for a CFG and its corresponding dominance relationships. 104

7.2 An overview for the execution of the proposed framework to select efficient target variables. 108

7.3 (Extended) CFG for Isqrt 111

7.4 Dominator tree for Isqrt 113

7.5 Dependency graph superimposed on dominator tree for Isqrt over its potential injection location set. 114

7.6 Variable Graph for Isqrt 115

7.7 Average error sensitivity distribution over all target programs for different variable selection methods. 128

7.7 Average error sensitivity distribution over all target programs for different variable selection methods. 129

7.7 Average error sensitivity distribution over all target programs for different variable selection methods. 130

8.1 Workflow for generating efficient fault injection points. 154

8.2 An overview of generated data set. 159

List of Tables

6.1 Register classificaiton scheme 70

6.2 Error resilience distribution of all fault models 77

6.3 Null hypotheis test results for fault model effect on error resilience 78

6.4 Estimated marginal means for error resilience of all fault models 79

6.5 Pairwise comparisons between mean for all fault models 80

6.6 Confidence interval of error resilience for all blocks 83

6.7 Error sensitivity distribution for different block locations under L_1C_1. 83

6.8 Error sensitivity distribution for different block locations under L_1C_2. 84

6.9 Error sensitivity distribution for different block locations under L_2C_1. 85

6.10 Spearman's rank-order correlations 92

6.11 Pearson product moment correlations 93

7.1 Number of target variables selected for the different target programs 117

7.2 Total number of fault injection experiments conducted over all target programs 118

7.3 Average error sensitivity distributions for different programs for L_1C_1 120

7.4 Average error sensitivity distributions for different programs for L_1C_2 121

7.5 Average error sensitivity distributions for different programs for L_1C_3 122

7.6 Average error sensitivity distributions for different programs for L_1C_4 123

7.7 Average error sensitivity distributions for different programs for L_2C_1 124

7.8 Average error sensitivity distributions for different programs for L_3C_1 125

7.9 Average error sensitivity distributions for different programs for L_4C_1 126

7.10 Average error sensitivity distributions for different programs for L_2C_2 127

8.1 The general form of a confusion matrix for binary classification. 146

8.2 (Double) Injection points efficiencies for naïve Bayes with no sampling 162

8.3 (Triple) Injection points efficiencies for naïve Bayes with no sampling 163

8.4 (Quadruple) Injection points efficiencies for naïve Bayes with no sampling 164

8.5 (Double) Injection points efficiencies for rule induction with no sampling . . . 164

8.6 (Triple) Injection points efficiencies for rule induction with no sampling . . . 165

8.7 (Quadruple) Injection points efficiencies for rule induction with no sampling . . . 165

8.8 (Double) Injection points efficiencies for decision tree induction with no sampling . . . 166

8.9 (Triple) Injection points efficiencies for decision tree induction with no sampling . . . 167

8.10 (Quadruple) Injection points efficiencies for decision tree induction with no sampling . . . 167

8.11 (Double) Injection points efficiencies for naïve Bayes with sampling 169

8.12 (Triple) Injection points efficiencies for naïve Bayes with sampling 169

8.13 (Quadruple) Injection points efficiencies for naïve Bayes with sampling . . . 170

8.14 (Double) Injection points efficiencies for rule induction with sampling . . . 170

8.15 (Triple) Injection points efficiencies for rule induction with sampling 170

8.16 (Quadruple) Injection points efficiencies for rule induction with sampling . . . 171

8.17 (Double) Injection points efficiencies for decision tree induction with sampling . . . 171

8.18 (Triple) Injection points efficiencies for decision tree induction with sampling 172

8.19 (Quadruple) Injection points efficiencies for decision tree induction with sampling 172

8.20 (Double) Injection points efficiencies for naïve Bayes with no sampling using full bit set 174

8.21 (Double) Injection points efficiencies for rule induction with no sampling using full bit set 174

8.22 (Double) Injection points efficiencies for decision tree induction with no sampling using full bit set 174

8.23 (Double) Injection points efficiencies for naïve Bayes with sampling using full bit set 175

8.24 (Double) Injection points efficiencies for rule induction with sampling using full bit set 175

8.25 (Double) Injection points efficiencies for decision tree induction with sampling using full bit set 176

List of Algorithms

7.1 Heuristic for Injection Location Selection (ILS) 105

7.2 Heuristic for Target Variables Selection 107

7.3 Algorithm to obtain a *variable* graph 116

CHAPTER 1

Introduction

Modern computer systems are now an inextricable part of the structure of modern societies. Part of these systems is often a computer control system, a *microcontroller*. A microcontroller is usually a self-contained system having a processor, memory and peripherals and can be used as an *embedded system*. Often microcontrollers are embedded in other machinery, such as automobiles, telephones, appliances, and peripherals for computer systems. Embedded systems range from portable devices, such as tablets and digital watches, to large stationary infrastructures, such as traffic lights, industrial process controllers and largely complex systems like hybrid vehicles, Magnetic Resonance Imaging (MRI) and avionics. Integral part of virtually all modern computer systems are *integrated circuits* (ICs). An IC is an electronic circuits on small plate of semiconductor device, usually silicon. As technology advances, IC scaling translates to a shrinkage in the feature size, reduction in supply voltage levels and increase in feature density and operation frequency. A microprocessor is an IC, or at most a few ICs, that contains all, or most of, the functions of a central processing unit (CPU) of a computer; and it is sometimes called a logic chip. Microprocessors are designed to perform binary, logic and arithmetic, operations that employs the usage of small number-holding areas called registers. Typical microprocessor operation include adding, subtracting, comparing two numbers, and writing and reading numbers to and from one area to another. These operations are the result of a set of instructions that are part of the microprocessor design. This set of instructions is called *instruction set architecture* (ISA), i.e.,

the ISA provides commands to the processor, to tell it what it requires to do. The ISA consist of different components which include (processor) registers. A register is one of a small set of data and it may hold an instruction, a storage address, or any kind of data (e.g., a bit sequence or individual characters). The low cost of IC made it possible for modern computer systems to pervade our everyday life. As such, this pervasive nature of these systems has increased our reliance upon such systems to provide correct and timely service.

However, these current hardware trends has exacerbated the unreliability of modern computer systems. As technology scales down, their sensitivity to their environment increases, as a result the probabilities of transient faults and soft-errors are increasing. A soft-error is an issue that causes a temporary condition in memory that alters stored data in an unintended way. This means, emerging technology are error prone to ionising radiation normally caused by low-energy neutrons coming from cosmic rays and alpha particles coming from packaging materials. Similarly, soft-error rate (SER) in logic circuits is increasing [111] and now comparable to the SER in unprotected memory, and the probability of multiple faults occurring in such devices is equally on the rise. Futhermore, these emerging technology are projected, in the near-future, to cause computer systems to expose hardware faults to the information-level, and to ensure that the software performs as specified [24, 28, 98, 114, 138]. Similarly, it has been demonstrated that many hardware faults manifest as multiple soft-errors [19]. All of these and the concomitant high cost associated with exclusively tolerating such faults at the hardware-level necessitate the design of software error resilient mechanisms and evaluating them under such hardware faults.

The type of fault tolerance to adopt and how to implement it is directly related and strongly dependent upon the underlying fault assumption. One of the major issues of designing fault tolerant systems is ensuring such systems meets their reliability requirements, that is, validating them. This is usually done with respect to the inputs, i.e., class(es) of faults, they were designed to cope

with [7, 124]. For a large part of applications, especially safety criical system, it is important to ascertain the coveragw of the fault inection process. There are other types of application that are not safety critical but may be prone to dervice degradation as a result of MBU, e.g., an image processing software rendering a blurry image. This motivates the need of multiple soft-errors model for designing and evaluating fault tolerant software systems.

1.1 Motivations

Research has shown assumptions about the types of faults that impact a software system and how they may affect the system are crucial in the design of a fault tolerant software system. Thus, dictating the relevant fault tolerance to implement. Similarly, such assumptions are relevant for the evaluation of the efficacy of the implemented fault tolerance mechanisms. As such, the emergence of multiple soft-errors and the eventuality of these errors increasing in the near future limits the accuracy of the traditional single fault model assumed during software dependability assessment. Similarly, the manifestation of soft-errors in several locations, including in logic circuits, constricts the applicability of the existing double faults model, emulating two soft-errors originating from a single location, assumed during software system dependability evaluation. Following, this motivates the need to define a multiple-locations multiple-corruptions fault model, to emulate multiple soft-errors occurring in several points, during dependable software assessment, which is the focus of the research presented in this thesis.

However, to assume a multiple-locations multiple-corruptions fault model for SWIFI three related issues arises; the problem of determining: (i) the injection location, i.e where to inject, (ii) the injection time, i.e., when to inject, and (iii) injection latency, i.e., how long to inject. The focus of the research presented in this thesis is on the problem of selecting the injection location, i.e., selecting

efficient injection points for multiple soft-error fault model. The contributions made in this thesis towards selecting efficient injection points for multiple soft-error are summarised in the next section and are based on the following thesis:

> **"There exists a computational feasible bits set to explore under multiple bit-flip faults that will induce a wider failure profile."**

1.2 Thesis Contributions

In general, this thesis works to address the challenges associated with multiple fault injection in terms of the type of faults to inject, where to inject them and the cost of fault injection in terms of number of experiments to perform. Specifically, this thesis contributes to the advancement of multiple fault injections by:

- Examining the problem of selecting efficient fault injection locations (in terms of inducing wider failure profile) in complex software and formalising this complexity. The formalisation is achieved by applying static analysis techniques and graph theory concepts on the software source or byte code. To formalise the complexity of selecting these locations, the work split the problem into two (i) injection location selection and (ii) target variable selection over all possible locations. The work proves both problems to be NP-complete.
- Proposing a novel fault model meant to be representative of emerging transient hardware faults that are due to hardware scaling and that may lead to multiple bit-flips in contrast to the single fault model traditionally assumed. This research studies the influence of such faults once converted into errors in software. The work extends the traditional model of single faults to multiple faults. The multiple faults are modelled as single

faults in combinations of several locations. The viability of the proposed fault model for software system validation is demonstrated in an extensive experimental fault injection analysis (more than 17 million individual experiments in thirteen embedded software modules). The research shows that, the novel fault model uncovers more vulnerabilities than the traditional single fault model, and causes more severe failures than a variant existing multiple fault model.

- Proposing an approach for selecting efficient fault injection (variable) locations in complex software, which take into account the relationship between program variables or states. The methodology is contrived to discern key variables for multiple-bits fault injections. The identification is done by applying static analysis techniques and graph theory concepts on the software source or byte code. To determine these locations, it provides two heuristic, the first to identify potential injection locations and the second, to identify minimal set of variables (in the potential injection location) to target. Further, this framework yields the multiple-locations multiple-corruptions fault model, $\mathrm{L}_n\mathrm{C}_m$. The work has also demonstrated the applicability of the framework and the validity of the $\mathrm{L}_n\mathrm{C}_m$ fault model on several case studies.

- Proposing and evaluating an approach to refine the faultload for multiple-bits fault injections by selecting a subset of fault injection points. This filtering is essential to reduce the fault space and cost (in terms of number) of multiple fault injection campaigns in embedded software modules. The proposed methodology is base on classification algorithms. In addition to the key bits identification, the methodology shows that fault injection done using these subset of key bits achieves similar efficiency as those done exhaustively with the entire set of bits.

1.3 Thesis Structure

This chapter has detailed the main motivations, contributions and thesis of the research to be presented in this thesis. The remainder of the thhesis will be structured as follows:

Chapter 2 provides basic dependability and fault tolerance validation concepts, principles and terminology that are central to the work presented in this dissertation. This account includes an overview of dependability attributes, threats and means, as well as discussion of fault assumptions and fault injection analysis, and the role they play in the design and assessment of fault tolerant software systems.

Chapter 3 describes the models under which the contributions made in this thesis have been developed, including details of the assumed model of the software systems, the fault models under which software dependability validation was considered and description of the fault injection tool and target programs adopted in the dependability evaluations.

Chapter 4 states the problem statements and provides a roadmap to the research presented in this thesis. The chapter elaborates on the potential problems of multiple fault injections addressed by the work presented in this thesis and maps this to the thesis contributions.

Chapter 5 analyses the complexity associated with the selection of efficient injection locations for injecting multiple soft-errors. Further, it formalises this complexity as two sub-problems and proves both problems to be NP-complete. The complexity is analysed in order to discover whether systematically obtaining an efficient tractable fault space for multiple soft-errors injections is possible.

Chapter 6 presents an empirical assessment of the limitations of the traditional single and current multiple fault models emulating software errors due single and

multiple hardware transient faults respectively, and proposes a variant multiple fault model for improving fault representativeness during dependable software validation. In addition, the chapter analyses the influence of the proposed fault model on software execution. To keep the fault injection experiments tractable, only double faults are considered.

Chapter 7 develops an approach for the careful selection of fault injection locations for each of the two associated problems presented in Chapter 5. The approach also considers the observations made in Chapter 6 to systematically identify injection locations for the $\mathrm{L}_n\mathrm{C}_m$ fault model.The approach is proposed in order to assist in streamlining the exponential fault space associated with multiple-soft-error injections with the goal to reveal as many software vulnerabilities as possibles. Following its development, the proposed approach is applied and the efficiency of the selected target variables in terms of uncovered vulnerabilities is measured.

Chapter 8 focuses on narrowing down the fault space for multiple soft-error injections. As such, it proposes an approach that applies data mining techniques to datasets obtained during multiple fault injection analysis. Following its development, the proposed approach is applied and the injection efficiency of the selected injection points is demonstrated.

Chapter 9 concludes the thesis with final remarks, summary of the research contributions and indication of applications area and future research directions.

CHAPTER 2

(Software) Dependability Concepts and Terminology

The prevalence of modern computer systems in all aspect of our daily lives, from consumer-oriented systems, such as automobiles and mobile phones, to high-end systems, such as nuclear power-plants and aircrafts etc, has prompted the increase in our dependence on such systems to render correct and timely service. Further, as technology advances, there is a concomitant increase of system functionality being defined in software and rise in frequency of faults and errors impacting these systems. Hence, it becomes crucial that software be dependable. In order to give an appropriate and consistent context presented in this thesis, this chapter describes and introduces the fundamentals and terminology in software dependability in general and topics that will be developed in subsequent chapters in particular.

2.1 The Fundamentals of Dependability

The fundamental concepts of dependability used throughout this disseration are adopted directly from the comprehensive compilation of concepts made by [7]. The *dependability* of a system is defined as the ability of the system to deliver service that can justifiably be trusted. The ability to avoid service failures that are more frequent and more severe than is acceptable is also defined as dependability of a system. Dependability of a system is characterised by a set of attributes, impaired by a set of threats and imparted by a set of means.

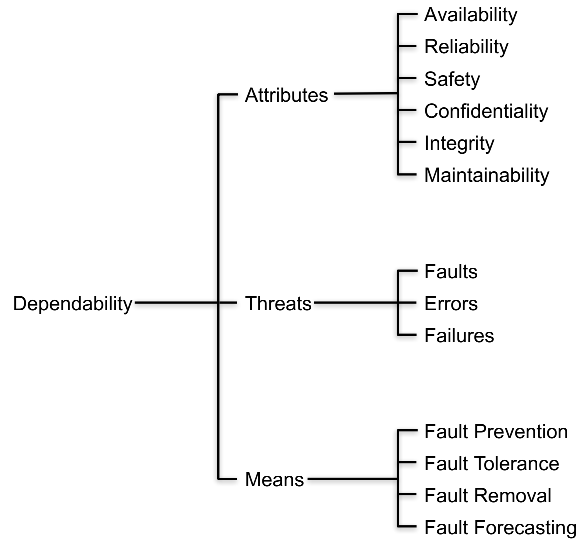


Figure 2.1: Dependability tree. [7]

2.1.1 Dependability Attributes

The dependability of a given system is characterised and profiled by the dependability attributes. These attributes are as follows:

Availability: The probability that the system is operational and providing its service at any given time is measured by availability. The higher the availability, the higher the likelihood that the system provides its service at the time that the service is requested. Or formally, availability is defined as a function of time representing the probability a service provided by a computer system is operating correctly and able to perform its designated function at a given time. Three frequently used availability terms are explained as follows:
Inherent availability, as seen by maintenance personnel, (excludes preventive maintenance outages, supply delays, and administrative delays) is defined as in Equation 2.1:

$$A_i = \frac{MTTF}{MTTF + MTTR} \tag{2.1}$$

where MTTF and MTTR represents the mean time to failure and the mean time to repair for the service respectively.

Achieved availability, as seen by the maintenance department, (includes both corrective and preventive maintenance but does not include supply delays and administrative delays) is defined as in Equation 2.2:

$$A_a = \frac{MTBF}{MTBF + MDT} \tag{2.2}$$

where MTBF and MDT represents the mean time between failure and the mean down time for the service respectively.

Operational availability, as seen by the user, is defined as in Equation 2.3:

$$A_o = \frac{uptime}{operatingcycle} \tag{2.3}$$

where operating cycle is the overall time period of operation being investigated.

Reliability: The probability that a system provides the service it was originally set to provide during a finite period of time is measured by reliability. This means that the higher the reliability, the higher the likelihood that the response given by a system is correct. Reliability is concerned with reducing the frequency of failures over a time interval and is a measure of the probability for failure-free operation during a given interval, i.e., it is a measure of success for a failure free operation. It is often expressed as in Equation 2.4:

$$R(t) = exp^{-\frac{t}{MTTF}} = 1 - exp^{-\lambda t} \tag{2.4}$$

where λ is constant failure rate.

Safety: The extent to which a system provides a service that is safe to its environment, i.e., it does not endanger the user, is measured by safety [7]. The safety measure may be higher than the reliability measure, in the sense that

the system may provide a service which was not originally intended, and this service may still be safe for users.

Confidentiality: The extent to which a system will allow those without sufficient privilege to obtain information that should not be made available is measured by confidentiality. The higher the confidentiality the higher the probability that it will not disclose undue information to non-authorised entities.

Integrity: The extent to which a system prevents alterations by unauthorised entity, or ensures an unauthorised entity does not prevent authorised modifications (including causing information interruption) by authorised entities is measured by integrity. Integrity is the absence of improper system alteration. The higher the integrity measure the higher the probability that a system will ensure that there is absence of improper systems alterations, with respect to withholding, modification and deletion of information.

Maintability: Maintainability is the measure of how long it takes to achieve (in terms of ease and speed) to restore outages to services provided by a system. Maintainability is the ability for a process to undergo modifications and repairs. The maintainability measurement is often the MTTR and a limit for the maximum repair time. Formally, maintainability is defined as a function of time representing the probability that a failed computer system will be repaired in t time or less. The maintainability attribute is conventionally denoted by $M(t)$. Where a constant rate of repair, μ, can be assumed, the maintainability of a system can be estimated by Equation 2.5:

$$M(t) = 1 - exp^{-\frac{t}{MTTR}} = 1 - exp^{-\mu t} \quad (2.5)$$

2.1.2 Dependability Threats

During the development and operation of a dependable system, events may occur that may impair the trustworthiness of the system by introducing *faults* into the system. A fault is a defect in system, i.e., a fault may be a software bug or effect of hardware fault. A system is said to provide correct service when the service is originally the one it set to provide, i.e., the service it provides complies with its functional specification. On the contrary, a system is said to provide incorrect service, i.e., a system failure is said to have occurred, when the service it provides differs from its functional specification. Typically, such system failure occurs due to the presence of threats to dependability. As shown in Figure 2.1, dependability threats are faults, errors and failures. However, the mere presence of faults is not sufficient to impair the dependability of the a system. A fault must become active, i.e., the part of the system the fault is located must be referenced in some way during the system execution. The activation of a fault may result in an *error* occurring. An error is a discrepancy between the intended behaviour of a system and its actual behaviour inside the system boundary, i.e., an error is erroneous state in the system. An active error may cause other errors to occur in the system. This process in called *error propagation*. Error propagation may result in system *failure* by preventing the system from providing correct services. That is failure occurs when error(s) propagate beyond the system boundary. i.e., if the error(s) become visible to the environment of the system.

The fault-error-failure error causality cycle is known as the fundamental chain an it is represented as follows:

$$fault \ \rightarrow \ error \rightarrow \ failure$$

The fundamental chain is recursive in nature. Thus what can be seen as a failure at one level of the system can be seen as a fault on the next level. Therefore,

these repetitive sequence leads to the definition extended chains of causality to represent the error propagation process, such as the following causality chain:

$$\cdots\ fault \xrightarrow{activation} error \xrightarrow{propagation} failure \xrightarrow{causation} fault\ \cdots$$

A fundamental capability of any dependable system is to limit the extent of error propagation. Given the nature of the fundamental chain it is possible to develop means to break these chains and thereby increase the dependability of a system.

2.1.3 Type of Faults

A fault can be classified into a hardware or a software fault according to where it occurs. A *hardware fault* is classified into a permanent, an intermittent, or a transient fault as indicated by the extent of its existence in a system (see Figure 2.2). This thesis focuses on hardware faults, which do not originate due to hardware damage and impact the execution flow of software and/or program. A *permanent fault* (stuck-at, stuck-open, and bridging faults) remains permanently in the system, an *intermittent fault* introduces repetitive broken data in a specific place because of hardware damage and a *transient fault* appears and disappears within a brief time. Permanent and intermittent faults occur because of inaccurate specifications, implementation mistakes, or component defects. A transient fault usually occurs because of internal and external noise.

The data errors that result from a hardware fault include hard- and soft-errors. *A hard-error* causes data corruption as a result of permanent and intermittent faults. A *soft-error* causes data corruption because of transient faults resulting from environmental disturbances, such as alpha particles or neutrons. As opposed to a hard error, a soft-error occurs under conditions where the device is not damaged. A soft-error can be divided into single and multiple bit-flips. A

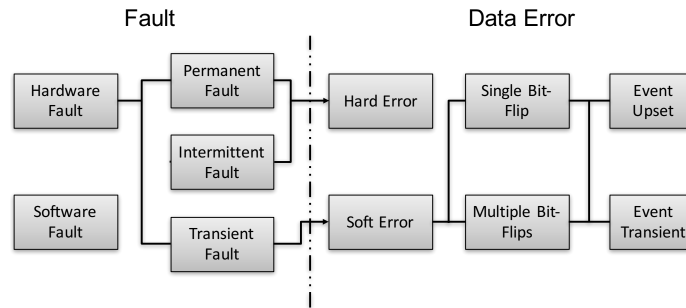


Figure 2.2: An overview of fault and error terminology focused on the transient hardware fault.

single bit-flip (SBF) consists of one bit-flip, and *multiple bit-flips* (MBF) consist of several bit-flips. Further, a bit-flip can be categorised into an event upset or an event transient, depending on where it manifests. An *event upset* manifests in storage element, e.g., in the latch or flip-flop, whereas an *event transient* occurs in combinational logic. Thus an SBF can be either be a Single Event Upset (SEU) or a Single Event Transient (SET); and an MBF is either a Multiple Event Upset (MEU) or a Multiple Event Transient (MET). It is common for the bits in a data word to not be physically adjacent, but interleaved with bits of other data words, i.e., bits in the same data word are physically number of bits apart from each other. This means that when an n-bit MBU occur, it may not affect bits in the same word. For it to translate into a data word MBU the following two condition must be true: (i) at least two of the failing bits in the MBU must belong to the same row, and (ii) the physical MBU must spread over more than the interleaved space. This interleaving architecture, typically makes physical MBUs manifest as data word SBUs [130]. For circuits protected with Error Correction Codes (ECC), such physical MBUs do not necessarily affect the performance of these circuits. Figure 2.3 shows an overview of MBU and Static Random Access Memory (SRAM) bits interleaving relationship. This thesis, thusly, focuses on MBUs that originate in the ISA registers (however, it does not consider errors in registers holding instructions).

The soft-error rate (SER) is defined as the occurrence rate of a soft-error in a

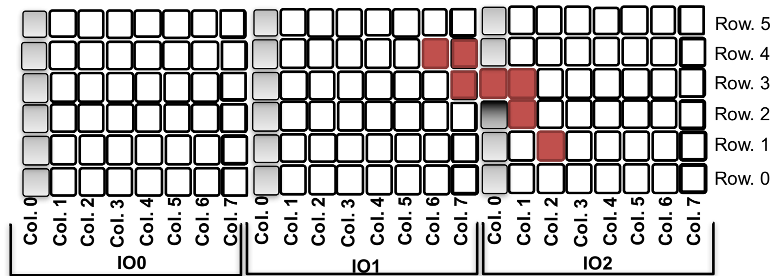


Figure 2.3: An overview of the relationshipship of an 8-bit MBU and a 3-bit data word with an 8-bit interleave.

device. The number of failures-in-time (FIT) or the mean time between failures (MTBF) are commonly used to express the SER.

In this thesis, an *impactful error* is considered to be those soft-errors that affect the software behaviour. Figure 2.4 depicts an overview of impactful errors and their propagation from the circuit level to the application level.

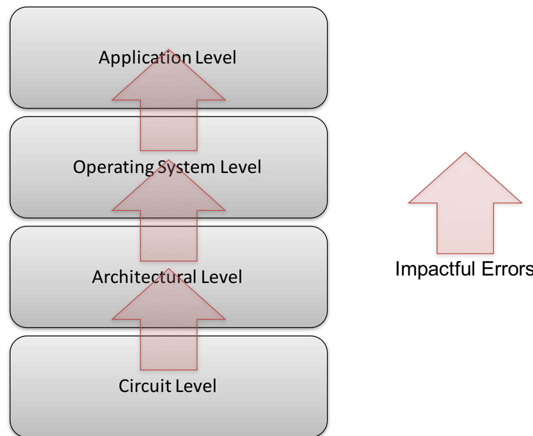


Figure 2.4: Block diagram of propagation of soft-error impacting software.

2.1.4 Dependability Means

When developing dependable systems, there are a number of means by which dependability can be achieved and analysed. As shown in Figure 2.1 and described as follows, the four dependability means are fault prevention, fault tolerance,

fault removal and fault forecasting:

Fault Prevention is the process of preventing faults being incorporated into a system. Fault prevention techniques focus on hindering and obstructing the occurrence, introduction and spread of faults. Established examples of such techniques include modular software design, software development methodologies and process quality assurance.

Fault Tolerance is the process of putting mechanisms in place that will allow a system to still deliver the required service in the presence of faults, although that service may be at a degraded level. Generally, such fault tolerance techniques focus on the recognition of an erroneous state in a system and restoring a suitably correct state, or at least a safe system state, following the occurrence of an error.

Fault Removal is the process of mitigating the number and seriousness of faults in a system. Fault removal techniques focus on reducing the number, likelihood of activation and wider consequences of faults in a computer system. Fault removal is generally a three stage process, where these steps are validation, diagnosis and system correction. Particularly, the validation stage focus on determining whether a system adheres to a set of defined properties, the diagnosis stage focus on identifying faults, which prevent these properties from being fulfilled and the system correction stage focus on modifying the system to allow the defined properties to be fulfilled.

Fault Forecasting is the process of predicting likely faults so that they can be removed or their effects can be circumvented. Fault forecasting techniques focus primarily on estimating the number, likelihood of activation and wider consequences of faults in a computer system. The fault forecasting process typically involves the identification, classification and analysis of modes by which a system can fail, as well as an evaluation of dependability attributes using probabilistic and analytical approaches. Fault injection analysis is a common

technique in usage when attempting to establish dependability measures and forecast fault proneness. Fault injection is a dependability validation approach whereby the behaviour of a system to the artificial insertion of faults or errors is analysed so that insights can be gained with respect to the dependability of the system

The contributions made in this thesis are generally related to the areas of fault tolerance and fault forecasting. In particular, the research presented in this thesis is focuses on improving the fault tolerance and fault forecasting mechanism dependability assessment and validation. More specifically, the research is concerned with demonstrating that the dependability assessment and validation process can be enhanced through the design of multiple fault model based on a set of candidate variables and candidate bits.

2.2 Fault Tolerance Validation

Fault tolerance techniques are not equally effective. The measure of efficacy of any given fault tolerance technique is called its coverage. The imperfections of fault tolerance, i.e., the lack of fault tolerance coverage, constitute a drastic impediment to the increase in dependability that can be achieved. Such imperfections of fault tolerance arise due to either:

- development faults that affect the fault tolerance mechanisms with respect to the fault assumptions specified during the development, the upshot of which lack of error and fault handling coverage, defined with respect to a class of errors or faults, (e.g., single errors, multiple errors etc), as the conditional probability that the technique is effective, given that the errors of faults have occurred,
- fault assumptions that are not representative of the fault that actually occur in operation, i.e the fault assumptions differ from the faults really

occurring in operation, resulting in a lack of fault assumption coverage, that can in turn be due to either (i) lack of failure mode coverage, i.e., the assumption on how failure occurs and (ii) lack of failure independence coverage, i.e., assuming components failure occur independently whereas they have a common failure trigger and vice vera.

Figure 2.5 summarises the relationship of fault tolerance of coverage. Fault tolerance coverage of a given technique is evaluated by means of validation techniques with respect to the fault tolerance assumptions the technique design is based on. There are several validation techniques, including formal methods, fault injection, and dependability analysis. Validation usually takes place at the end of the development cycle, and looks at the complete system as opposed to verification, which focuses on smaller sub-systems. Verification is the process of checking that the system conforms to its specification

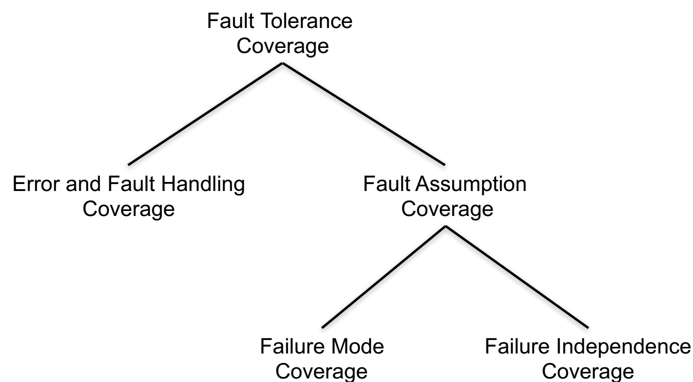


Figure 2.5: Overview of fault tolerance coverage [7].

2.2.1 Formal Method

Formal methods are concerned with the use of mathematical and logical techniques to express, investigate, and analyse the specification, design, documentation, and behaviour of both hardware and software. Formal methods are originally used as verification techniques, however, they are now being employed

in validation of fault tolerance techniques. For example Ayache et al. [8] defines a methodological framework applicable to the early life cycle phases of fault-tolerant systems engineering. The framework focuses on the verification of fault tolerance properties using model-based formalisms. Lecocke et al. [86] describes an approach to fault tolerant design and implementation that uses a formal model to automatically generate fault detection and response methods. The approach is designed for resource-constrained embedded systems with high reliability requirements such as manned or critical space assets. Fey et al. [44] propose the use of formal methods to assess the robustness of a digital circuit with respect to transient faults. The formal model uses a fixed bound in time to cope with the complexity of the underlying sequential equivalence check.

2.2.2 Fault Injection

As has been mentioned in the previous chapter, fault injection is the intentional activation of faults by either hardware or software techniques to observe the system operation under the effect of the fault. Fault injection is adopted to evaluate the dependability of a system. Fault injection may be used to determine vulnerable parts of a system in order to design, assess and improve fault tolerant systems. The fault injection system interacts with the target system for fault activation, process control, and fault analysis. Figure 2.6 depicts fundamental fault injection workflow and Figure 2.7 summarises a basic fault tolerance validation process. Fault injections techniques can be classified as hardware-based, software-based, simulation-based and emulation-based. These are briefly described in the following sections.

Hardware-Based Fault Injection

Hardware-implemented fault injection is also called physical fault injection because faults are actually injected into the physical hardware [5, 21, 46, 74].

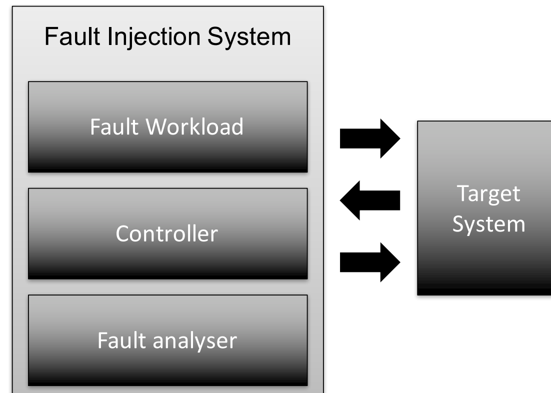


Figure 2.6: An overview of a basic fault injection environment.

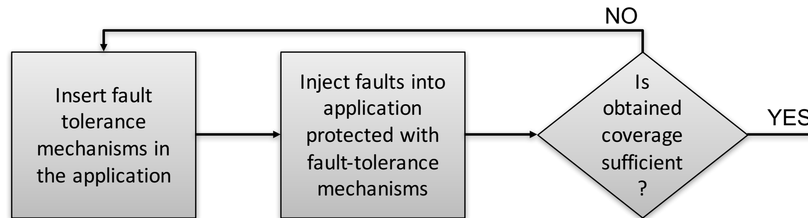


Figure 2.7: An overview of fault tolerance validation.

Hardware fault injection introduces a direct stimulus at the pins or socket. The circuit is tested using the change in the operating power or temperature or the external shocks that cause transient errors. The testing speed is fast owing to the real-time fault injection structure. By directly changing the environment, a wide range of circuits can be evaluated through these disturbances. However, its processes are difficult to monitor and control because the exact moment when a fault is injected by the disturbance is not known. A drawback of hardware-based fault injection is there exists a possibility of damaging the target system as actual circuits cannot be restored after testing.

Software-Based Fault Injection

Traditionally, software-based fault injection involves the modification of the software executing on the system under analysis in order to provide the capabil-

ity to modify the system state according to the programmer's modelling view of the system. This is done as a possible way to assess the consequences of software bugs. However, software-based fault injection have been extended to assess not just software bugs, but other faults that can impact the operation of system at the application level. All types of faults may be injected, from register and memory faults, to dropped or replicated network packets, to erroneous error conditions and flags to transient hardware faults. These faults may be injected into simulations of complex systems where the interactions are understood though not the details of implementation, or they may be injected into operating systems to examine the effects. Fault injection is a widely used technique in software dependability evaluation, e.g., [58, 73, 103, 156, 163]. The work presented in this thesis falls under the area of SWIFI.

Simulation-based and Emulation-Based Fault Injection

Simulation-based fault injection is concerned with the construction of a simulation model of the system under analysis, including a detailed simulation model of the processor in use [29, 45, 95, 135, 145]. This means that the errors or failures of the simulated system occur according to predetermined distribution. The simulation models are designed using a hardware description language such as the Very high speed integrated circuit Hardware Description Language (VHDL). Faults are injected into VHDL models of the design and activated by a set of input patterns. Emulation-based fault injection are designed to cope with the time limitations imposed by simulation and to take into account the effect due to the circuit environment in the application, in system emulation using hardware prototyping on FPGA-based logic emulation systems [20, 94].

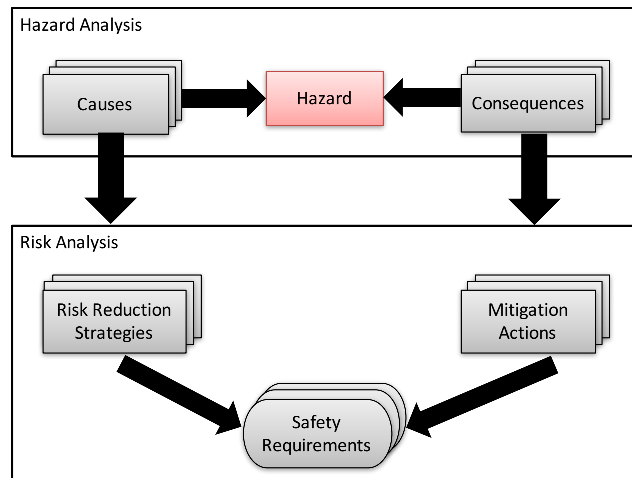


Figure 2.8: An overview of dependability analysis.

2.2.3 Dependability Analysis

Dependability analysis is the process identifying hazards and then proposing methods that reduces the risk of the hazard occurring. Dependability is categorised into hazard analysis and risk analysis. Hazard analysis is the process of recognising hazards that may arise from a system or its environment, documenting their unwanted consequences and analysing their potential causes. Hazard analysis involves using guidelines to identify hazards, their root causes, and possible countermeasures. Risk analysis takes hazard analysis further by identifying the possible consequences of each hazard and their probability of occurring. Dependability analysis is being used in the validation of fault-tolerant, e.g., [116, 140, 153, 167, 168]. Figure 2.8 summarises the basic dependability analysis workflow.

CHAPTER 3

System and Faults Models and Target Systems

To be able to perform dependable software validation, the software system model along with fault model considered has to be specified. This chapter describes the software model assumed in the development of the contributions made in this thesis, and the fault models under which they were assessed. This chapter also introduces all target systems together with their associated input set, system failure modes, software system instrumentation procedures and dependability validation techniques used to evaluate and illustrate the approaches presented in this thesis.

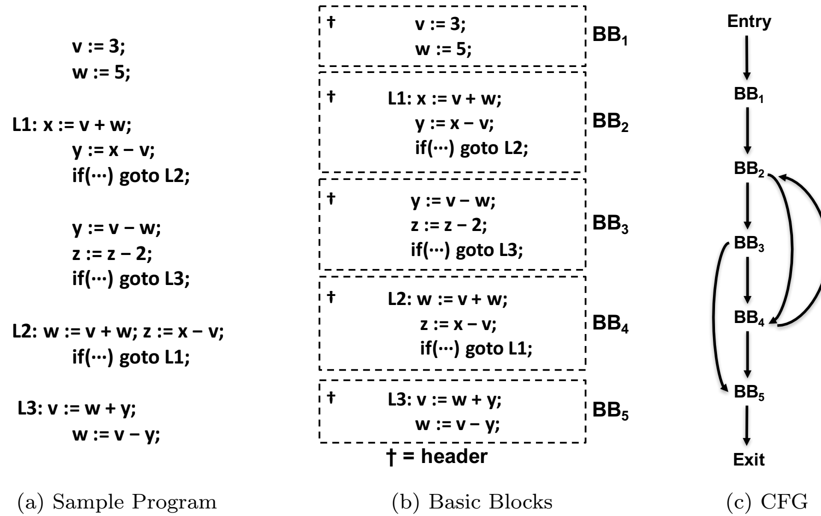
3.1 System Model

This thesis considers modular software, i.e., software consisting of a number of discrete software functions called modules, that interact to deliver the requisite functionality. A module is considered as a generalised white-box, having possibly multiple inputs and outputs and whose codebase is available.

Modules communicate with each other in some specified way using different forms of signalling, such as, shared memory, parameter passing etc. A software module performs computations using the inputs received on its input channels to generate outputs, which are then placed on the requisite output channels. At the lowest level, such a module may be a procedure or a function and a process at the highest level. A software consists of such modules that interact via signals.

Signals can originate (or end) from hardware or from another module. Such type of software is common place nowadays, and can be seen in many different application areas, such as embedded systems. In this thesis, henceforth, modules is used interchagebly with software systems, unless otherwise specified, and a software system is modelled as an *extended control flow graph* (extended-CFG).

A control flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. In a CFG, a node in the graph represents a sequence of statements called basic block (or block for short), i.e. a straight-line piece of code with branching only allowed at the end. Directed edges are used to represent possible transfer of control. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow exits.



(a) Sample Program (b) Basic Blocks (c) CFG

Figure 3.1: Example of basic blocks for a dummy program and its corresponding CFG.

The first statement in a basic block is a header, the target of any branch is a header, and the statement following any branch is a header. Thus each basic block is consist of a header at the entry and the ensuing sequence of statements between it and the next header. In a CFG, there exists a directed edge from

basic-block1, BB_1 to basic-block2, BB_2, i.e $BB_1 \longrightarrow BB_2$, if: (i) there exist a branch from the last statement in BB_1 to the header of BB_2 and/or (ii) BB_1 does not end in an unconditional loop and it immediately precedes BB_2. There is at most one edge for any given direction between BB_1 and BB_2, i.e, not more than one edge exists for $BB_1 \longrightarrow BB_2$. There is an edge From *Entry* to the initial basic block, there is an edge from each final basic block to *Exit*. Figure 3.1 shows a sample program code, and an overview of its basic blocks and CFG.

In this thesis, an extended-CFG is obtained from its CFG by ensuring each node does not contain a program variable that is depended upon another program variable within the said node, i.e., no self-loop exists in any given block. In the next section, an extended-CFG is formally defined.

3.1.1 Extended-CFG for a Program

An Extended CFG for a program P is a labeled weighted directed graph $G_P = \langle V, v_0, A, W, \Phi \rangle$, where

- V: is a set of vertices, with each vertex $v \in V$ representing a block in P, and each block represents a sequence of consecutive instructions or statements in P.
- v_0: is the *root vertex*, representing the starting block in P. It has an in-degree of 0.
- A: is a set of arcs (u, v), where $u, v \in V$. An arc exists between u and v if execution of block u can directly lead to block u.
- W: is a function $W : A \to \mathbb{N}$, that defines a weight for each arc (u, v) in G_P. In this thesis, it is assumed that assume the weight to represent the number of steps or statements of the block$v$.

- In a given block, whenever there is a data dependency between two variables, the block is split into two such that the dependency is across blocks. Thus, there is no data dependency within a block. This is done based on the assumption that error propagation occurs across blocks, rather than within a block.

Extended-CFG under Multiple Soft-Error

The extended-CFG used for Multiple-bits errors have this additional property:

- Φ: is a function $\Phi : V \to \mathbb{B}$, where $\mathbb{B}$ is a function that assigns a boolean value to each vertex $v \in V$. The vertex is assigned a value 1 if it is a potentially vulnerable block, 0 otherwise (details are provided in Chapter 5.3).

Notation: The set of paths in G is denoted by ρ_G and the set of paths between two vertices $u, u' \in V$ by $\rho_G^{u,u'}$. Given a path $\rho = u \cdot v \ldots v' \cdot u'$, then $\hat{\rho} = \{a | a \in v \ldots v'\}$. The length of a path ρ in G, denoted by $Length(\rho)$, is given by $\sum_{(u,v) \in \rho} W(u, v)$.

3.2 Fault Model

This thesis considers the transient hardware faults originating at the transistor level, that ultimately impact on the software modules. As previously mentioned, these faults are aggravated by current hardware trends. These faults impact on the program state by altering the content of CPU registers and memory, and through the process of error propagation causes errors [7] to exist in the software system. These errors are usually mimicked by injecting bit-flip errors[1]

[1] In this thesis bit-flip errors are taken to mean the same as corruptions, and from this point, both would be used interchangeably unless specified otherwise.

in registers and memory. The general assumption is any number of bit-flip errors may occur in any number of locations. This thesis considers single bit-flips and multiple bit-flips errors occurring. As mentioned, this thesis focuses on soft-errors in register locations only.

3.2.1 Single Fault

Traditionally a single fault assumption has been used for fault injection analysis. This means in a given location a single bit is flipped in a single execution of the program. Research has shown multiple bits errors [2] occurring in the field as single-cell, single-row, single-column, multiple-rows, multiple-columns or chip-wide errors [51, 97, 150]. This pinpoints the need for considering multiple-bits errors in software dependable validation. This thesis adopts the Single Bit-Flip or L_1C_1 fault model as baseline for evaluating the efficiency of the adopted multiple-bits errors models.

3.2.2 Multiple Faults

In consideration of the potential of multiple soft-errors affecting the running software, research have begun studying the impact of double-bits errors on software for dependability evaluation [9]. In [163], the double-bits fault model has also been shown to mimic the presence of software bugs. The mentioned research focused on a version of double-bits errors occurring within a single location. In this thesis, this fault model is modelled as two bits flipped within a single location, and referred to as model as Double Bit-Flips or L_1C_2 (which is a specific case of the L_nC_m fault model) . Lu et. al. [103] adopted the double-bits errors to show the applicability of their fault injection tool. This thesis adopts the L_1C_2 fault model to evaluate the viability of the proposed variant double-bits models in terms of its ability to induce programs to fail differently. This thesis

[2] In this thesis bits errors is short for bit-flips errors.

adopts a second pattern of double hardware faults occurring as single faults in a pair of locations. This thesis models this fault model as double single bit-flip errors in two different locations, and this is refered to as L_2C_1 (which is also a specific case of the L_nC_m fault model). The thesis proposes this model in order to ascertain the need to adopt it for the purposes of dependable software validation. The model assumes in any run of the program only two errors can occur, as such it selects two locations and flip one bit in each. It should be mentioned that the work presented in [103] tested the applicability of their fault injection tool with both the L_1C_2 and L_2C_1 fault models, and their work post dates that presented in [2] which serves as the basis of some of the work presented in this thesis.

This thesis generalises the double faults model to allow multiple faults to be introduced in a single run instead of two. This model assumes any number of errors can occur in a single execution of a program, as such several locations are selected and a minimum of a single bit is flipped in each location. This new multiple faults model is referred to as Multiple Locations Multiple Multiple Corruptions (L_nC_m, where n is the number of injection locations and m the maximum number of faults to inject in each location) fault model.

From the extended-CFG perspective of a program, it means that (i) several variables in a given block can be corrupted, (ii) several blocks can be corrupted, with a single variable being corrupted in each block or (iii) several variables being corrupted in several blocks of the program. As this thesis focuses on capturing the impact of multiple hardware faults on a program, it is important to (i) determine the location (block) where the fault will be injected and (ii) determine the variables in which the faults will be injected into. At one extreme, the entry block (root of the control flow graph) can be chosen and all variables in that block being selected as target variables. At the other extreme, every variable within every block can be target variables. However, the computational cost of validation will be prohibitive.

This thesis only evaluates L_nC_m having a maximum of four faults. That is, in addition to the aforementioned double-bits errors, L_1C_2 and L_2C_1,the thesis also assumes triple faults models, where (i) three bits are flipped in a single location (L_1C_3) and (ii) three single bits are flipped in three different locations (L_3C_1), and quadruple bits errors, where (i) fours bits are flipped in a single location (L_1C_4), (ii) four single bits are flipped in four different locations (L_4C_1) and (iii) double bits are flipped in a pair of locations (L_2C_2).

3.3 Target Systems

An overview of each target program used in this thesis is provided in following sections. From this point onwards, program is used interchangeably with module and software system (or system for short), unless it is otherwise stated.

3.3.1 Flight Control

Flight Control is a safety-critical system, Mathwork's implementation of a flight control system for the longitudinal motion of an aircraft [109]. First order linear approximations of the aircraft and actuator behaviour are connected to an analog flight control design that uses the pilot's stick pitch command as the set point for the aircraft's pitch attitude and uses aircraft pitch angle and pitch rate to determine commands. To perturb the system, a simplified Dryden wind gust model is incorporated. Within the flight control system, two programs were used for instrumentation:

- **Derivatives**: This program updates derivatives for the root system
- **Step**: This program updates the model step

The input data for these programs is a pilot frequency in rads/secs.

3.3.2 SUSAN (Smallest Univalue Segment Assimilating - Nucleus)

SUSAN is an image recognition package, developed for noise filtering and for recognising corners and edges in Magnetic Resonance Image (MRI) of the brain [152]. SUSAN is available as a self-contained C program from [53]. SUSAN is also available as a program in the automotive package of the MiBench suite [53]. It is typical of a real world program that would be employed for a vision based quality assurance application. For example, it may be used for digitally processing images to determine the position of edges and/or corners therein for guidance of unmanned vehicle. In SUSAN, three different programs were targeted:

- **Corners**: Performs corner (two feature) detection.
- **Edges**: Performs edge (one feature) detection.
- **Smoothing**: Performs structure preserving noise reduction (noise filtering).

The input data for SUSAN are set of Netpbm grayscale image format (PGM).

3.3.3 MiBench Suite

MiBench [53] suite consists of a benchmark suite targeting embedded processing environments. In the MiBench suite, programs from these packages were instrumented:

Automotive and Industrial Control Package

Benchmarks in the Automotive and Industrial Control (Automotive for short) package are intended to show use of embedded processors in embedded control

systems. These processors require performance in basic math abilities, bit manipulation, data input/output and simple data organisation. Typical real applications for these programs are air bag controllers, engine performance monitors and sensor systems.

Two benchmarks from the Automotive package were chosen to be used for fault injection. These programs are selected because they perform simple, usually necessary, mathematical calculations that mostly do not have dedicated support in embedded systems. For example, cubic function solving and integer square root are all necessary calculations for calculating road speed or other vector values.

- **Cubic Equation Calculator (Cubic)**: This program calculates the square root of the input.
- **Square Root Calculator (Isqrt)**: This program calculates the roots of a cubic equation using floating point arithmetic implemented in the software.

The input data for these programs is a fixed set of constants.

Telecommunications Package

With the explosive growth of the Internet, many portable consumer devices are integrating wireless communication. These benchmarks consist of voice encoding and decoding algorithms, frequency analysis and a checksum algorithm. The programs chosen from the Telecommunications package are as follows:

- **CRC**: This program performs a 32-bit Cyclic Redundancy Check (CRC) on a file. CRC checks are often used to detect errors in data transmission. The input data for CRC are speech samples.

- **FFT**: This programs performs a Fast Fourier Transform and its inverse transform on an array of data. Fourier transforms are used in digital signal processing to find the frequencies contained in a given input signal. The input data is a polynomial function with pseudorandom amplitude and frequency sinusoidal components.

Network Package

The benchmarks in this package represent embedded processors in network devices like switches and routers. The work done by these embedded processors involves shortest path calculations, tree and table lookups and data input/output. The algorithms used to show the networking category include finding a shortest path in a graph and creating and searching a Patricia trie data structure. The following programs are chosen in the Network package:

- **Dijkstra**: The Dijkstra program constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm. Dijkstra's algorithm is a well known solution to the shortest path problem and completes in $O(n^2)$ time.
- **Patricia**: A Patricia trie is a data structure used in place of full trees with very sparse leaf nodes. Branches with only a single leaf are collapsed upwards in the trie to reduce traversal time at the expense of code complexity. Usually, Patricia tries are used to represent routing tables in network applications. The input data for this benchmark is a list of IP traffic from a highly active web server for a two hour period. The IP numbers are disguised. The following algorithms are targeted from Patricia:
  - **insert**: This program performs insertion operations, i.e., it adds new element(s) in the trie.

  - **remove**: This program performs deletions operations by deleting the specified element(s) from the trie.
  - **search**: This program performs lookup operations in order to determine if an element exists in the trie.

Security Packages

As the Internet continues to gain popularity in e-commerce activities, the importance of data security is also increasing. The Security package includes several common algorithms for data encryption, decryption and hashing. One program is chosen to be targeted, rijndael, the new Advanced Encryption Standard (AES). AES is based on a design principle known as a substitution-permutation network, combination of both substitution and permutation, and is fast in both software and hardware.

- **Rijndael**: Rijndael was selected as the National Institute of Standards and Technologies Advanced Encryption Standard (AES). It is a block cipher with the option of 128-, 192-, and 256-bit keys and blocks. The input data sets are ASCII text file of articles found online. The following algorithms are targeted:
  - **encfile**: This program performs encryption operations, i.e, it encrypts the input data.
  - **decfile**: This program performs decryption operations on an encrypted input.

Each program have nine inputs, of three varying sizes.

3.4 Fault Injection Analysis

This thesis uses the LLVM Fault Injection Tool (LLFI) [156] to introduce faults into target programs. LLFI is a LLVM-based fault injection tool that works at the LLVM [84] compiler's intermediate representation (IR) level.

3.4.1 LLVM

Low Level Virtual Machine (LLVM) [84] is a compiler infrastructure designed as a set of reusable libraries with well-defined interfaces for program analysis and optimisation. LLVM consists of (i) a front-end to translate program code written in a high-level language such as C/C++ to an intermediate representation and (ii) a backend to translate the intermediate representation (IR) into machine code for specific platforms. The IR is a low-level programming language similar to assembly. The IR is a strongly typed RISC instruction set which abstracts away details of the target. It can be transformed by multiple optimisation passes before being converted to the machine code by the backend. The LLVM intermediate representation is a typed language in which source-level constructs can be easily represented. It preserves the variable and function names, making source mapping feasible. Further, LLVM has extensive support for program analysis and transformations which makes it easier to study the effect of fault injection at a higher level than assembly language.

3.4.2 LLVM Fault Injection (LLFI) Tool

On the account that LLFI target programs at the IR code, it allows fault-injections to be performed at specific program points and into specific instructions. The effect can then be easily tracked back to the source code. LLFI supports various fault injection customisations, and enables tracing the propagation of the resulting error among instructions in the program [156].

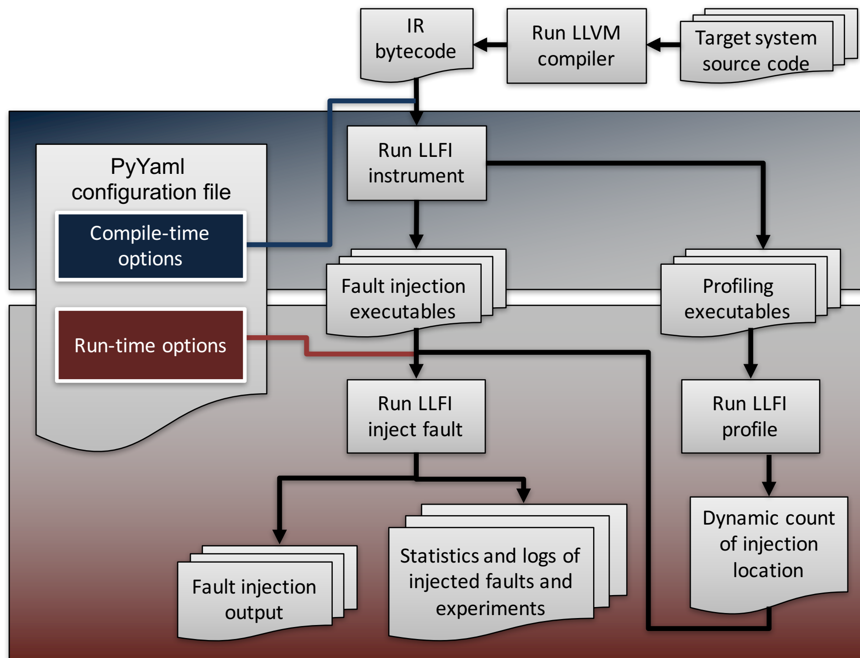


Figure 3.2: LLFI workflow [156].

LLFI Workflow and System Instrumentation

Figure 3.2 shows the working of LLFI, which consists of three compulsory stages.

First stage, entails running LLFI instrument. In this step, LLFI takes the program byte-code as input, and applies a configuration instructions specified the fault injection configuration script written in PyYaml format. The configuration file contains both (i) compile-time options, including injection location and register to target in the specified injection location, and (ii) the run-time options, consisting of the fault model, i.e the type of fault to inject, number of experiments to run and time out definitions. The LLFI instruments target injection locations with calls to fault injection functions. Call-backs functions are also instrumented for profiling. Running the LLFI instruments produces as output fault-injection and profiling executables, a dynamic count of injection locations is also logged.

Secondly, the LLFI profile runs the profiling executables produced in Step 1 in order to create a golden run of the target program, where a golden run is a reproducible fault free execution of the system. This is done in provide a baseline for comparisons with fault injection executions. This step also create more fault injection setup text files.

Thirdly, the fault injection executables created in Step 1 is executed at runtime, and LLFI randomly selects one runtime instance of the instrumented instructions to trigger the fault injection function and inject into the selected instruction operand value leveraging the information in setup files created in Step 2. Because hardware faults occur randomly at runtime, LLFI picks a random instruction from the set of all dynamically executed instructions at runtime to inject into. This is possible because the fault injection function is invoked at runtime, and can hence choose which invocation of an instruction to inject into. The output of the fault-injector is the fault injection experiments, including program output, log and stat file. The log files captures execution information including program exceptions and system crashes etc, while the stat files stores execution information such as injected fault type, injection location etc.

Further, by instrumenting the program once with the set of all fault-injection functions, and injecting the fault at runtime, LLFI ensures that the same executable file (with the instrumentation in it) is used in all the fault injection runs. Finally, this method makes it unnecessary to recompile the code for each fault injection. Other work on high-level fault injection has followed a similar approach [16, 59, 73]. The result is then logged (i.e., where the fault was injected, what type) and the program allowed to continue. The final output of the program is also logged. This requires the profiling step to have completed successfully and the corresponding stat file to have been created. However, when any compile-time options is changed, the process has to start from Step 1.

In an optional final step, LLFI traces the propagation of faults in the IR code

and allows visualisation how the IR code are mapped to the source code. Trace is collected only if specified in Step 1. This generates an execution trace after the profiling and fault-injection steps, for each fault injected. The traces can be compared to identify how the fault propagated.

3.4.3 Failure Scheme

To better understand the failure profile of the targeted program, the outcome of the fault injection experiments are categorised, using a purpose-built tool, as follows:

- **No Impact:** If the program execution terminates normally, and the output produced is identical to the output produced by the fault-free execution of the program, the outcome is labelled as *No Impact*. This fault-free execution of the program is referred to as *Golden Run*.
- **Exception Failure (Exception):** The outcome is classified as an *Exception Failure*, if the program execution encounters an unexpected error that does not result in the program crashing or hanging, i.e., the program terminates normally but does not produce any output.
- **Silent Data Corruption (SDC):** If the program execution terminates normally, but produced an output that deviates from that produces by the golden runs, the outcome is classified as *Silent Data Corruption (SDC)*.
- **Time Out Failure (Time Out):** The outcome is classified as a *Time Out* failure, if the program execution hangs, i.e. fails to terminate within predefined time. This time is arbitrary set to be approximately 15 times larger than the execution time of the golden run.
- **Crash Failure (Crash):** If the program execution is terminated unexpectedly when it encounters an unexpected error, the outcome is classified as a *Crash* failure.

CHAPTER 4

Problem Statements

Fault tolerance mechanisms have traditionally been validated through the use of software-implemented fault injection (SWIFI). Several SWIFI frameworks exist, however most of them are based on single-fault assumption, i.e., they assume that a single fault will occur in any execution run of the system. During the validation process, this assumption translates into a single fault being injected into an execution of the system, overlooking any potential interactions between simultaneous independent faults. Such a fault assumption has become less appropriate and limited as (i) software systems containing more than a single fault are more often the norm rather than the exception [14], (ii) current safety standards require the consideration of multi-point faults [62] and (iii) it has been shown that simultaneous fault injections can efficiently detect robustness vulnerabilities [163].

In general, during fault injection, (i) the type of faults to inject, (ii) which variable to inject in (referred to as the injection location) and (iii) the time at which the fault injection occurs, are usually considered. Under the single soft-error assumption, i.e., single bit-flip, the fault space is linearly large (in the size of the word or register and in the number of variables), making exhaustive fault injection feasible. Generally, when multiple fault injections are considered: (i) several variables can be targeted at any given point, and/or (ii) several different locations can be target at a given time. When multiple soft-errors are considered, important challenges arises, the most important ones being to deal

with the *exponential* size (in the number of variables, variables combinations and the bit-positions) of the fault space. To make the multiple fault injection process efficient, it is important to inject fault combinations that are likely to convey more information and uncover more vulnerabilities, i.e., cause the system to fail. Moreover, the values to inject at the injection location is also an important dimension to consider for multiple faults assumption. Thus, the following challenges are identified in support of the thesis statement:

> **“There exists a computational feasible bits to explore under multiple bit-flip faults that will induce a wider failure profile.”**

4.1 Selecting Potential Injection Blocks Locations

> Does a set of locations (blocks) exists which will be suitable candidate blocks for multiple faults injections? How can these blocks (potential injection locations) be identified?

Having addressed this problem, it is important to discern the best combination of variables to target. Thus following problem needs to be considered.

4.2 Identifying Candidate Variables to Target

> Does a set of candidate variables exists within the set of potential injection locations that will be suitable for injecting multiple faults? How can they be identified? To what extent is dependability validation improved? That is, what is the probability of uncovering vulnerabilities?

To address this problem, the nature of interactions between variables needs to be understood and thus the associated problem of error propagation when multiple faults are considered for injections arises.

4.2.1 Error Propagation Masking

> Does an activated fault nullify the effect of a previously activated fault? Does an activated fault prevent another fault from being activated? Can faults that may potentially *mask* another be identified?

If multiple faults are injected, an activated fault f_1 can potentially mask another fault (or error) f_2 if, for instance, either f_1 prevented f_2 from being activated or f_1 cancelled the effect of f_2. The following illustrations elaborate on error masking:

Prevention of Fault Activation: Considering the following program code, the following is a typical problem that can occur during multiple fault injections:

```
2    Z = X + Y;
4    if Z ≤ v2 then
6        Y = v2 − X;
8    else
10       Y = v1 − X;
12   c = Y;
```

Assuming that at a given point during the program's execution at Line 2, $v_2 = 7$, $X = 1$ and $Y = 0$; in this instance, in the absence of any fault, the value of Z will be 1, the program execution will true for the branching condition in Line 4 and thus the value of Y would be updated at Line 6, and in turn C will assume the value of Y at Line 12. Also, assuming each variable is 4-bit long, and Y is targeted at Line 2. Any fault injection that causes the state of Z to assume a value between 0 to 7 will be masked, this means the injected fault will not

change the execution flow, and the state of Y and C would be updated with the correct value at Line 6 and Line 12, respectively. Further, any injected fault that changes the state of Z to become a value in the range of 8 to 15 would cause the execution flow to change and consequently the state of Y and C will be incorrectly updated at Line 10 and Line 12 respectively, which may likely lead to some sort of system failure. This means, some injections would be wasteful, i.e., certain injections exercise the system in the same way, and faults targeted at locations whose state may be used to determine branching conditions may not likely get activated.

However, assuming Y is instead targeted after the program execution have exited the loop at Line 12, the state of C will be updated with the corrupted Y value. Thus, targeting Y at that location ensures the fault will be activated, and subsequently increasing the probability of causing a system failure.

Nullification of the Effect of Activated Fault: Considering another dummy program, another typical problem associated with multiple fault injections could occur as follows:

```
2    a1 = a2;
4    b1 = getB();
5    ⋮    ⋮
7    E = a1 − b1;
8    ⋮    ⋮
```

Supposing that during a given program execution $a_2 = 2$ at Line 2 and $b_1 = 0$ at Line 7, this means during a fault free execution a_1 and E will assume the values 2, at Line 2 and 2, at Line 7, respectively. Assuming also, that double single faults are injected if: location a_2 is targeted at Line 2 and location b_1 is targeted at Line 7. Supposing each location is 4-bit long, and at Line 2, the fault corrupted the first bit position in a_2 thus changing its state from 2 to 3. Consequently, at Line 7, E will be computed as 3. Assuming the second fault is

activated at this location and the first position is corrupted, thereby changing the state of E from 3 to 2. Thus, the second activated fault in E will mask the first corruption in a_2, i.e., the effect of the second fault cancels the effect of the first fault.

4.2.2 Error Propagation Amplification

> Does an activated fault increase the effect of a previously activated fault? Would flipping a single bit be more impactful than flipping multiple bits in a given location? Can faults that may potentially *amplify* the effect of each other or another be identified?

Another ambiguity that can occur during multiple fault injections is reduction of the impact of an error when another fault gets activated. However, when the effect of an activated fault is increased by a another fault being activated then *amplification* has occurred. If multiple faults are injected, an activated fault f_2 can potentially reduce the impact of another error f_1, if, for instance, f_1 leads to a corrupted value in location l that deviates with a large difference from the supposedd correct value l at that point, and f_2 updates the state of l with a corrupt value that deviates with the a small difference of the correct value of l. Considering the following sample code illustrates the concept of amplification:

```
1   ⋮    ⋮
3   a := getA();
5   c := getC();
7   b := c + a;
8   ⋮    ⋮
```

Assuming all the variables are 4-bits long integers, and at a given execution cycle of the program, $getA()$ returns a value 4 at Line 3. In the absence of fault, an error the value of a at that point will be 4, and let $c = 5$ at Line 5. In a fault

free execution of the program $b = 9$ at Line 7. If, a fault is activated at Line 3, and it corrupts the state of a by making $a = 0$, without the activation of any other, the corruption will propagate to b at Line 7 ($b = 5$, deviating from its fault-free value by -4). Supposing a second fault is activated at Line 7 that led c to have a corrupted value of 8, with both faults activated at Line 7 ($b = 8$, just 1 less than its fault-free value). In this scenario, the activation of the second fault has reduced the deviation introduced by the first fault activated, thus the second fault did not amplify the effects of the second fault, and thereby reducing the likelihood of system failure. However, supposing at Line 5, the state of c is corrupted with the value 13 rather than 8, this will result in b assuming a corrupted value of 13 (4 more than its fault-free value) instead of 8. In this second scenario, the second error have increased the effect of the first error, i.e., amplification has occurred, and the likelihood of a system failure is increased.

Another type of amplification effect that may potentially occur is when multiple bit-flips are considered within a single location. Still considering the preceding source code, maintaining the assumptions of each variable being 4-bits long integers and in a given point of a fault-free execution of the program $getA()$ returns a value of 4 at Line 3 and $getC()$ returns a value of 5 at Line 5, then $b = 9$ at Line 7. Considering a at Line 3 to be the only target location, where any number of faults can be injected, i.e., any number of bits can be flipped. In a fault-free execution, a will be represented as 0100. If, the first bit from left is flipped, a will become 1100 (12), when the error propagates to b at Line 7, b will be 17 (8 more than its fault-free value). Supposing, three faults are introduced into a in the first, second and third bit-positions from right; a then becomes 0011 (3), and when the error propagates to b at Line 7, b will be 8 (just -1 less than its fault-free value). In this scenario, injecting multiple faults is less likely to induce a system failure than the injecting single fault. On the hand, if a single fault is introduced in the first bit-position from the right, ($a = 0101$ (5)), and multiple fault are injected in the first, second and third bit-positions from the

left, ($a = 1010$ (10)), then at Line 7, $b = 10$ (just 1 less than its fault-free value) and $b = 15$ (6 more than its fault-free value), respectively. In this scenario, the impact of multiple faults is more likely to induce a system failure. This implies that in the case of multiple faults within a single location, amplification may or may not occur when multiple bits are flipped as opposed to flipping a single bit in the said location.

Having identified a set of good injection candidates, it is desirable to have efficient bit-positions to perturb. As it has been highlighted above, some bit-positions may not be good injection points. Moreover, the fault space remains exponentially large in terms of variable and bit-position combinations, $2^n \cdot 2^m$, where n is the number of candidate variables and m the number of bits. Thus, the final problem statement to be considered is:

4.3 Selecting Choice Bit-Positions

> Does an efficient set fault injection point exists? If so, how can the most suitable points for injecting multiple faults be identified? How well do the proposed injection points uncover vulnerabilities? That is, how high is their probability of inducing a system failure?

These problem statements have guided the work that is now presented in this thesis, and hopefully, some light can be shed upon these problems. How these problems maps to the main thesis contributions are briefly described in the next section.

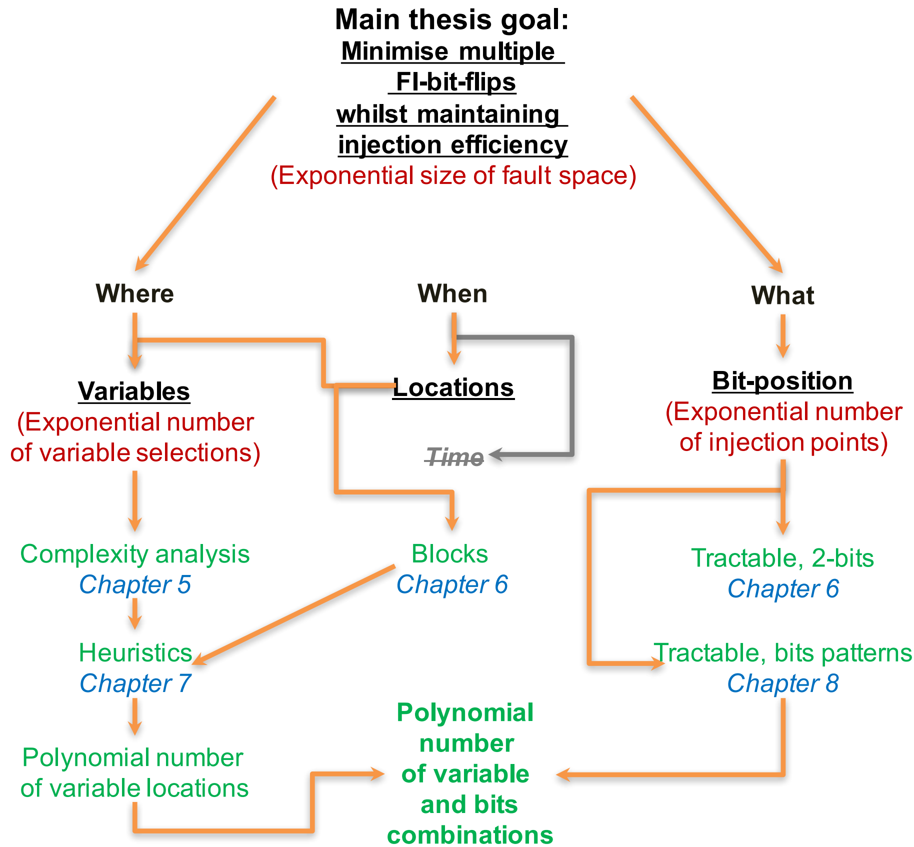


Figure 4.1: An overview of the thesis contributions.

4.4 Roadmap of Thesis Statement

This thesis seeks to address the problems associated with multiple soft-error injections in general, and in particular it aims to minimise the cost of fault injection campaigns while maximising injection efficiency[1], in terms of uncovering vulnerabilities. Thus, the main objective of the thesis is to discover a set of bit-positions that exists within a set of variable locations which when flipped will uncover as much vulnerabilities as if the entire bits set is flipped. The major problems associated with multiple faults injections include determining *what to inject, where to inject, when to inject and how long to inject.* What to inject,

[1]This thesis considers injection efficiency of a location as its ability to induce a failure when a fault or an error is injected into it.

where to inject and when to inject are the hurdles this thesis must overcome in order to achieve its main goal. Figure 4.1 summaries the objective of the thesis, the problem statements associated with this goal, the chapters that address these problems and the outcome of addressing the problems.

Knowing *where* to inject will help in determining suitable injection locations that will potentially induce a high proportion of system failures. Finding these candidate locations requires understanding how program variables interact with one another and how these interactions ultimately effect the program execution. These issues are elaborated as Problem Statements 4.2.1 and 4.2.2, addressing these problems involves understanding a program's control flow and data dependencies. Understanding these relationship highlights that there exists an intuition that will guide in choosing candidate variables. This, also highlights the challenge of dealing with an exponential space, in the number of variables to consider and in the number of possible variable combinations. To address this challenge, first, the associated complexity of identifying candidate variables needs to be understood. Chapter 5 addresses some of the issues stated in Problem Statements 4.1 and 4.2, by analysing and formalising the complexity of finding suitable variable candidates. To fully tackle the exponential space challenge, *when dimension* of injecting faults needs to be concomitantly investigated. Thus, Chapter 6 investigated the effect of locations in terms of variable placement hierarchy in both the control and execution flows of the program. However, the aspect of when to inject the fault, in terms of relative time spent in executing the program, i.e., faults being triggered by timing-triggers, is a component not considered in this thesis. Leveraging, the information uncovered in Chapters 5 and 6, the remainder issues stated in Problem Statements 4.1 and 4.2 are addressed in Chapter 7 by creating a framework, to first identify locations that can potentially improve fault injection efficiency, and then to choose the subset of candidate variables that would circumvent the issues stated in Problem Statements 4.2.1 and 4.2.2. The solutions, up to this point, have narrowed

down the fault injection space by making the variable locations space polynomial. However, the fault space remains exponentially large in terms of number of variables and bits combinations, this brings about the issues stated in Problem Statement 4.3 and highlights the *what dimension* of injecting faults. To address this challenge, this aspect of the combinatorial space of variables and bit-position needs to be made tractable by identifying choice bit-positions that can potentially uncover as much vulnerabilities as if the entire set of bit-positions (in terms of variables-bits combinations) have been used. The need to identify bit-position is implicitly highlighted in Problem Statement 4.2.2, and Chapter 6, attempts to achieve bits space tractability by considering only double bits, albeit, bits and variables choice are not systematically determined. Chapter 8 addresses this issue by applying data mining algorithms to fault injection data generated by targeting the identified candidate variables in Chapter 7. Thus, achieving the objective of this thesis of minimising multiple bit-flips fault injections and maximising injection efficiency, in terms inducing as much failures as using entire bits combinations.

CHAPTER 5

Towards Selecting Locations for Multiple Soft-Errors Injection

As emphasised in earlier chapters, the main objective of the work presented in this thesis is to find and evaluate new approaches for reducing exponential size of multiple fault injection point space whilst selecting those points that will induce a similar profile as in exhaustive fault injection with all points. Thus far, importance on how soft-error manifests in a system including when and where they appear and how long they remain in the system has been shown. However, before the duration and occurrence-timing dimensions can be considered, it is crucial to determine when, where and how they occur. It should be mentioned that the duration and timing dimensions are not investigated in this thesis.

In Chapter 4, potential problems associated with multiple fault injections are stated. One of these problems highlights the importance of the location dimension of multiple fault injections, in terms of selecting locations that would enhance the efficiency of injections whilst minimising the overall injections. However, very few works have addressed this complexity problem in a systematic way [9, 103, 163]. For example, Winter et al. [163], to handle this large fault space, target the variables at the input interface, and either randomly chooses a small subset from the fault space (for fuzzing fault type) or flips only a small number bits (for the bit-flip fault type), to keep the fault space polynomial in size. Also to circumvent the exponential fault space for multiple bit-flips fault type, randomly chooses a small subset of locations and then, either flips

only a small number of bits in one location [1, 2, 9, 103] or flips one bit in a small number of locations [1, 2, 103]. Thus, current state-of- the-art techniques in SWIFI-based software validation using MBFs either flip a small number of bits (e.g., [9, 156, 163]) or use a small number of random values (for fuzzing fault type) to assign to chosen variables [163] or target a small number of variables [156]. However, the efficiency of MBFs in uncovering vulnerabilities is better than that of fuzzing[1] [163], indicating the need for a systematic way to determine the multiple-bits combinations to flip during fault injection. The research in [163] focuses of injecting software bugs and the works in [1, 2, 9, 103] focus of injecting soft-errors. Some of work presented in this thesis is based on the work in [1, 2].

This generally means that when multiple fault injections is considered, at one extreme, similar to the single fault model, a fault is injected only in one location and let it effect propagate through the program execution. And at the other extreme, a fault is injected in each location, which may likely cancel earlier injections and is also computationally intractable. This means, there is a need to inject fault in a subset of locations, as injection in every single location is infeasible and injection in a single location may not produce accurate results.

However, some intuition may need to be considered when selecting candidate injection locations as injecting in some location may also lead to the undesirable occurrence of a later injection cancelling the effect of an earlier injection. This occurs if, for example, a fault f_1 leads to a corrupted value in a location, and by activation of a different fault f_2 that particular location is never referenced, the effect of f_1*'s* activation never becomes visible although it would have if f_2 had not been activated. Or if, for instance, a fault f_1 leads to a corrupted value in a location l_1, and by activation of a different fault f_2 resets the corrupted value in l_1 (See Chapter 4).

[1] Fuzzing, also fuzz testing, is a software testing technique used to discover coding errors and security loopholes in software, operating systems or networks by inputting massive amounts of random data, called fuzz, to the system in an attempt to make it crash.

This necessitates selecting meaningful candidate variables for multiple faults injections. Following, to systematically select such efficient variables, the following problems have to be addressed: (i) choosing the locations in which faults can be injected and (ii) choosing the variables in which faults will be injected. In order to do so, it is necessary to understand the complexity associated with achieving these task. And this complexity can be understood by applying computational complexity theory concepts to analyse the complexity.

5.1 Basic concepts of Computational Complexity Theory

Computational complexity theory is a subfield of theoretical computer science that is concerned with the study of the intrinsic complexity of computational tasks. Its most important goals include determining the complexity of any well-defined task and obtaining an understanding of the relations between various computational concepts, i.e., it focuses on classifying computational problems according to their inherent difficulty, and relating those classes to each other. A computational problem is a problem that may be solved by systematic application of mathematical steps, such as an algorithm or a heuristic. A problem is regarded as inherently difficult if its solution requires significant resources in terms of whatever the algorithm used, including time and space [147]. Consider the following instances: (i) Given two natural numbers n and m, are they relatively prime, i.e., do n and m posses greatest common divisor 1?, (ii) Given a propositional formula Φ, does it have a satisfying assignment? And (iii) Given a chess board of size $n \times n$, does white have a winning strategy if play is started from a given initial position? These problems are equally difficult from the perspective of classical computability theory in the sense that they are all effectively decidable. Yet they still seem to differ significantly in practical difficulty. For having been supplied with a pair of numbers $m > n > 0$, it is possible

to determine their relative primality by a method which requires a number of steps proportional to $log(n)$, e.g., Euclid's algorithm. On the other hand, all known methods for solving the latter two problems necesitate a 'brute force' search through a large space of cases which increase at least exponentially in the size of the problem instance.

Complexity theory attempts to emphasise such distinctions by proposing a formal criterion for what it means for a mathematical problem to be feasibly decidable, i.e., that it can be solved by a conventional Turing machine in a number of steps which is proportional to a polynomial function of the size of its input. The class of problems with this property is known as **P**, *polynomial time*, and includes the first of the three problems described above. P can be formally shown to be distinct from certain other classes such as **EXP**, *exponential time*, which includes the third problem from above. The second problem from above belongs to a complexity class known as **NP**, *non-deterministic polynomial time*, consisting of those problems which can be correctly decided by some computation of a non-deterministic Turing machine in a number of steps which is a polynomial function of the size of its input. A famous conjecture, often regarded as the most fundamental in all of theoretical computer science, states that P is also properly contained in NP, i.e., $P \subseteq NP$. Many complexity classes are defined using the concept of a reducibility and completeness.

5.1.1 Reducibility, NP-hardness and NP-completeness

A reduction is the mapping of one problem into another problem, i.e., a transformation of one problem into another problem. It captures the informal notion of a problem being at least as difficult as another problem. For example, if a problem X can be solved using an algorithm for solving Y, that denotes that solving Y is at least as difficult as solving X, and that X is reducible to Y. There are many different types of reductions, based on the method of reduc-

tion, such as Cook reductions [25], Karp reductions [75], and the bound on the complexity of reductions, such as polynomial-time reductions or log-space reductions. The most common reduction in usage is polynomial-time reduction, i.e., the reduction process takes polynomial time. For instance, the problem of squaring an integer can be reduced to the problem of multiplying two integers. This means an algorithm for multiplying two integers can be used to square an integer. Indeed, this can be done by giving the same input to both inputs of the multiplication algorithm. Thus, it can be seen that squaring is not harder than multiplication, since squaring is reducible to multiplication. This motivates the concept of NP-hardness, i.e the concept of a problem being hard for a complexity class. A problem X is understood to be hard for a complexity class C, if every problem in C, is reducible to X. This means that an algorithm for solving X can be used to solve every problem in C and solving X is at least as difficult as solving any problem in C. The set of problems that are hard for NP is the set of *NP-hard* problems, i.e., a problem is NP-hard if it is a member of NP and if an algorithm for solving it can be translated into one for solving a known NP-hard problem. If a problem Y is a member of a complexity class C and hard for C, i.e., all problems in C are reducible to Y, then Y is said to be complete for C. The completeness of Y for C may thus be understood as demonstrating that Y is representative of the most difficult problems in C. The set of problems that are complete for NP is the set of *NP-complete* problems, i.e., a problem is NP-complete, if it is a member of NP and is it also in NP-hard.

This chapter, focuses on capturing the complexity of (i) choosing the locations in which faults can be injected and (ii) choosing the variables in which faults will be injected. This chapter formalises each problem as an optimisation problem, and shows them to be NP-complete. Specifically, the problem of injection selection is formalised and its intractability is shown by proving the The *minimum vertex cover* (MVC) problem is reducible to it (See Section 5.3.1). A *vertex cover* of a graph G can simply be thought of as a set S of vertices of G such that every

edge of G is incident to at least one member of S, i.e., each edge of G has at least one member of S as an endpoint. The vertex set of a graph is therefore always a vertex cover. The smallest possible vertex cover for a given graph G is known as an MVS, and its size is called the vertex cover number, denoted $\tau(G)$ [121]. Finding an MVC of a general graph is a classical NP-complete decision problem in computational complexity theory [75]. Formalised also, is the problem of target variables selection and its complexity is proven by showing that the *minimum dominating set* (MDS) is reducible to it (See Section 5.4.1). For a graph G, the dominating set of G is a subset S of the vertex set $V(G)$, such that every vertex in V that is in S is adjacent to at least one member of S. The domination number $\gamma(G)$ is the number of vertices in a smallest dominating set for G. The MDS problem concerns finding a minimum such S and it is a known NP-complete problem [48] . The dominating set problem concerns testing whether $\gamma(G) \leq K$ for a given graph G and input K; it is a classical NP-complete decision problem in computational complexity theory [49].

It should be mentioned that this chapter does not consider the number of corruptions that can occur within a given variable or location. However, this aspect of the research will be addressed in later chapters.

5.2 Selecting Locations for Mulitple Fault Injections

SWIFI is an experimental technique that has been extensively used to evaluate the robustness and dependability of software systems, e.g., [58, 73, 156, 163]. Software systems that have been experimented on range from safety-critical embedded systems, e.g., [58, 159] to non-critical operating system's components [40, 137]. Most of them assume a single bit upset to emulate hardware faults that occur. With single bit upsets, the size of the fault space varies

linearly with the word or register size, making exhaustive bit-flips possible, e.g., [58, 67, 91].

However, the single fault assumption rules out multiple faults and their possible interactions during execution. It has been shown that multiple fault injections can be very effective in detecting software vulnerabilities [163]. Furthermore, research has found that many hardware faults manifest as multiple bit-flips in the program, and hence traditional single bit-flip injection may not be sufficient to model these faults [19, 97]. The impact of overlooking multiple fault injections is wide ranging: the software may appear to be more dependable than it is [163] and hence, the error handling mechanisms designed for the software may only have limited coverage or efficiency. In spite of these important problem, few works are addressing mulitple fault injections because of concomitant computational cost associated with it. To make multiple fault injection tractable it becomes neccessary to identify a minimal set of locations that can be corrupted such that wider system failure occurs. This neccesitates understanding the complexity associated with selecting efficient injection points. The remainder of this chapter, shows the complexity analysis of selecting injection points for multiple fault injections.

5.3 Injection Location Selection (ILS)

For multiple fault injections to work, first, a set of potential locations for injections need to identified. However, selecting such a set of potential locations at which to inject faults is very difficult. This chapter formalises the problem, analyses its complexity and shows it to be NP-complete by mapping the ILS problem into *minimum vertex cover* problem. The following section (Section 5.3.1) shows the formalisation and the proofs.

5.3.1 Complexity Analysis of ILS

Given a program P as its CFG[2] $G_P = \langle V, v_0, A, W, \Phi \rangle$ of P, where Φ represents the labelling (or tagging) of locations as vulnerable or not. The *amplification* factor is denoted by $\mathbb{A}$. The amplification factor captures the rate at which faults are injected in a given run. For a high amplification factor, then the rate is high, i.e., the number of blocks between successive fault injection locations is very small. On the other hand, for a low amplification factor, the number of blocks is high. For example, for the highest amplification factor, a fault can be injected in every single block. On the other hand, for the smallest amplification factor, a fault is injected in a single block (as in traditional fault injection). Thus, it is considered that the length of a path between two successive potential injection locations as a measure of amplification, with the shorter the length, the higher the amplification. Thus, for multiple fault injections, the objective is to select the smallest number of potential locations at which faults can be injected that satisfies $\mathbb{A}$.

Now, depending on the inputs to the program P, execution may follow different paths in P. Since it is difficult to know which execution path the system will follow, it is imperative then that the set of potential locations spans every possible execution path of P, i.e., every possible execution path of P has several selected potential locations.

From a graph perspective, the injection location selection problem is as follows: select a set $V^l \subseteq V$ such that, for any two vertices $u, u' \in V^l$, the length of the *longest* path between u and u' that does not go through a distinct node $u'' \in V^l$ does not exceed $\mathbb{A}$. The set V^l captures the set of possible injection locations. Thus, $\mathbb{A}$ has to be set such that $\mathbb{A}$ is at least equal to twice the longest distance between two successive potential locations as it may be possible that one such location is overlooked when choosing target variables.

[2]CFG, here means an extended-CFG and henceforth CFG is used interchangeably with extended-CFG unless specified otherwise.

Formally, the problem is defined as follows:

Definition 5.1 (Injection Location Selection (ILS)). *Given a CFG* $G_P = \langle U_P, u_0, A, W, \Phi \rangle$ *of program* P*, an amplification factor* $\mathbb{A}$*, and a positive integer* $K^P \leq |V|$*, then does there exist* $U^l \subseteq U_P$ *such that:*

- $|U^l| \leq K^P$
- $v_0 \in U^l$
- $\forall u \in U^l : \Phi(u) = 1$
- $\forall u, u' \in U^l :$
  $\forall p \in \rho_G^{u,u'} :$
  $U^l \cap \hat{p} = \emptyset :$
  $Length(p) \leq \mathbb{A}$

Lemma 5.1.1 (ILS). *ILS is in NP.*

Proof. To prove this, the correctness of the solution set U^l is required to be verified in polynomial-time. So, given an instance of ILS and a solution set U^l, correctness of U^l is verified as follows: The first three conditions can be trivially verified.

For the fourth condition, it is needed to verify that, from any vertex $u \in U^l$, all paths originating from u will contain another vertex $u' \in U^l$ with distance at most $\mathbb{A}$ away. This is done follows: First, a node $u \in U^l$ is selected, and a spanning tree of depth $\mathbb{A}$ rooted at u constructed, by doing a depth-first traversal on G. This tree is denoted by U^t. Now, given graph U^t, it is required to verify whether for every path p originating from u and ending at a leaf has at least one vertex $u' \in U^l$. If the answer is negative, then U^l is not a solution for ILS. On there other hand, if the answer is true for U, then the process is repeated for all other vertices $u \in U^l$. The complexity of this verification procedure is $O(|U|^2)$. □

Lemma 5.1.2 (NP-hardness). *ILS is NP-hard.*

Proof. To prove this, a known NP-hard problem, the MVC problem [48] is reduced to ILS. First, the MVC problem is defined:

MVC: Given a graph $G = (V, E)$ and a positive integer K, find a set $V' \subseteq V$ such that:

- $|V'| \leq K$
- $\forall (u, v) \in E : u \in V' \vee v \in V'$

With this definition of MVC, the work now develops the mapping between ILS and MVC.

Mapping

It is assumed that graph for MVC has a vertex with in-degree 0, denoted by v_0, which do not change the complexity of MVC.

- $U_P = V$
- $u_0 = v_0$
- $A = E$
- $W(a) = 1, \forall a \in A$
- $\mathbb{A} = 2$
- $K^P = K$
- $\Phi(u) = 1, \forall u \in U$

Reduction

It is now imperative to show that a solution to MVC exists if and only if a solution of ILS exists.

($\Rightarrow$) Let $V' \subseteq V$ be a solution to MVC with graph $G = (V, E)$. Let U^l be a solution to the instance of ILS as defined under the mapping, for graph $G' = (U_P, u_0, A, W, \Phi)$, with amplification factor $\mathbb{A} = 2$, such that

$U^l = V'$. It has been shown that this solution U^l is valid for ILS. First, since V' is a solution to MVC and $U^l = V'$, then $|U^l| \leq K$. Secondly, since $\Phi(u) = 1, \forall u \in U$, then $\forall v \in U^l : \Phi(v) = 1$. Finally, for every edge $(m, n) \in E \Rightarrow m \in V' \vee n \in V'$. For the case where only one disjunct is satisfied, i.e., $(m \in V')$, then the maximum distance to another vertex $u \in V'$ is at most 2, which is equal to the amplification factor $\mathbb{A}$. In the case of both disjuncts being satisfied, then the distance between 2 vertices in V' is 1, which is less than $\mathbb{A}$. Hence, the maximum distance between any pair of vertices in V' is at most 2, thus not violating $\mathbb{A}$. Since $U^l = V'$, V' is a solution to ILS.

($\Leftarrow$) Let $U^l \subseteq U_P$ be a solution to the instance of ILS defined under the previous mapping for graph $G' = (U_P, u_0, A, W, \Phi)$ with amplification factor $\mathbb{A} = 2$. Now, it is assumed a graph $G = (V, E)$ for MVC, with solution set $V' \subseteq V$. It has been shown that V' is a solution to MVC, when $V' = U^l$. Given that U^l is a solution to ILS, $(|U^l| \leq K^P) \wedge V' = U^l \Rightarrow |V'| \leq K^P$. Secondly, since the maximum distance between any pair of vertices (m, n) on the same path in G' that are in V' is at most 2, then it means that either $(m, n) \in E$ or $\exists k : (m, k) \in E \wedge (k, n) \in E$. Since $V' = U^l$, U^l is a solution to ILS

□

Theorem 5.1.1 (NP-completeness)**.** *ILS is NP-complete.*

Proof. The proof follows trivially from Lemmas 5.1.1 and 5.1.2. □

It has thus been shown that the selection of fault injection locations is NP-complete. To circumvent this high complexity, a heuristic that select a set of potential locations is proposed and studied in later chapter of this thesis.

5.4 Target Variable Selection (TVS)

Once the set of potential locations has been identified, it is then necessary to determine the set of variables into which faults will be actually injected. Specifically, there may be locations at which no fault will be injected and other locations where several faults may be injected.

The challenge in selecting target variables set from the fact that when a variable u is overlooked, then it means either that a variable v on which it depends has been selected (and selecting u will override the effect of propagating error from v to u) or a variable w that depends on v has been selected. Thus, the decision of selecting a variable is not a local one. This problem then is very similar to the problem of generating dominating sets. Thus, this chapter proceeds to prove that the problem of target variables selection (TVS) is NP-complete.

5.4.1 Complexity Analysis of TVS

The problem of target variables selection is formally defined as an optimisation problem.

Definition 5.2 (Target Variables Selection (TVS))**.** *Given a graph* $G_P^D = (U, A, U^0, W, L)$*, where* $U^0 \subseteq U$*, a positive integer* N*, a positive integer* $\mathbb{A}$*, does there exist a set* $U^V \subseteq U$ *such that:*

- $|U^V| \leq N$
- $\forall u, u' \in U^V$:
  $\forall p \in \rho_G^{u,u'}$:
  $U^V \cap \hat{p} = \emptyset$:
  $length(p) \leq \mathbb{A}$

It should be observed here that the amplification factor is carried over from the injection location selection problem (ILS). Specifically, given that no two

successive potential locations are no more than distance $\mathbb{A}$ apart, the selection of target variables should not violate this requirement.

Lemma 5.2.1 (NP membership)**.** *TVS is in NP.*

Proof. To prove this, the correctness of the set U^v is shown in polynomial-time. So, given an instance of TVS as described and a solution set U^v, the verification is performed as follows: The first condition is trivially verified. For the second condition, it is required to verify that, from any vertex $u \in U^v$, all paths originating from u will contain another vertex $u' \in U^v$ with distance at most $\mathbb{A}$ away. This is done as follows: It first select a node $u \in U^v$, and construct a spanning tree of depth $\mathbb{A}$, rooted at u, by doing a depth-first traversal on G. This tree is denoted by U^t. Now, given graph U^t , it is needed to verify whether for every path p originating from u and ending at a leaf has at least one vertex $u' \in U^v$. If the answer is negative, then U^v is not a solution for TVS. On the other hand, if the answer is true for U , then the process is repeated for all other vertices $u \in U^v$. The complexity of this verification procedure is $O(|U|^2)$. □

Lemma 5.2.2 (NP-hardness)**.** *TVS is NP-hard.*

Proof. To prove this, MDS problem [48] is reduced to the TVS problem. Before defining the MDS problem, it is denoted by U^1, the set of vertices adjacent to a node $u \in V$. Then, the MDS problem is defined as follows:

MDS: Given a graph $G = (V, E)$, a positive integer K, find a set $V' \subseteq V$ such that

- $|V'| \leq K$
- $\forall u \notin V' : \exists v \in U^1 : v \in V'$

With this definition of MDS, now the mapping between MDS and TVS is developed.

Mapping

It is assume that the graph for MDS has a set of vertices with in-degree 0, denoted by V_0, which does not affect the complexity of MDS.

- $U = V$
- $U^0 = V^0$
- $A = E$
- $N = K$
- $W(a) = 1, \forall a \in A$
- $\mathbb{A} = 3$.

Reduction

It now has to be to shown that a solution to MDS exists if and only if a solution of TVS exists.

($\Rightarrow$) Let $V' \subseteq V$ be a solution to MDS with graph $G = (V, E)$. Let U^v be a solution to the instance of TVS as defined under the mapping, for graph $G' = (U, U^0, A, W)$ with amplification factor $\mathbb{A} = 3$, such that $U^v = V'$. It is shown that this solution U^v is valid for TVS. First, since V' is a solution to MDS and $U^v = V'$, then $|U^v| \leq K$. Secondly, for $\forall n \notin V' : \exists m \in N^1 : m \in V'$. Now, assume there is a edge $(p, q) \in A$ such that $p, q \notin V'$. Then, there are two extreme cases: (i) if $\exists m \in P^1 : m \neq q \wedge m \notin Q^1 : m \in V'$ and $\exists n \in Q^1 : n \neq p \wedge n \notin P^1 : n \in V'$, then the distance between nodes m and n is at most 3, satisfying $\mathbb{A}$, and (ii) $\exists m \in P^1 : m \neq q$ and $\exists n \in Q^1 : n \neq p$, then if $m = n$, then the distance is 0, which is less than $\mathbb{A}$. Hence, the maximum distance between any pair of vertices in V' is at most 3, thus not violating $\mathbb{A}$. Since $U^l = V'$, V' is a solution to TVS.

($\Leftarrow$) Let U^v be a solution to the instance of TVS as defined under the mapping, for graph $G' = (U, U^0, A, W)$ with amplification factor $\mathbb{A} = 3$. Let $V' \subseteq V$ be a solution to MDS with graph $G = (V, E)$ such that $V' = U^v$. Now,

since U^v is a solution for the defined instance of TVS and $|U^v| \leq k$, then $|V'| \leq k$. Further, since no node $n \in V'$ is more than distance $= 3$ from some other node $m \in V'$, then it means that there is at most two nodes between nodes m and n that are not in V', i.e., $\forall p \notin V' : \exists q \in P^1 : q \in V'$. Since $V' = U^v$, then U^v is a solution to MDS.

□

Theorem 5.2.1 (NP-completeness)**.** *TVS is NP-complete.*

Proof. The proof follows trivially from Lemmas 5.2.1 and 5.2.2. □

To circumvent the complexity of target variables selection, a heuristic is developed and investigated later in this thesis.

5.5 Summary and Conclusions

This chapter investigates the complexity associated with selecting efficient locations for injecting multiple soft-errors. To understand the problem associated with selecting the efficient injection locations, the problem is split into two: (i) injection location selection and (ii) target variables selection at the potential locations. Following, this thesis formalises each problem and proved both problems to be NP-complete using graph theory concepts.

To prove the NP-completeness of the injection location selection, first, the problem is formally defined as a graph optimisation problem, ILS using the CFG for programs. Second, the correctness of the ILS problem is verified in polynomial-time with complexity of $O(|U|^2)$. Thus proving ILS to be in NP. Third, MVC problem is mapped and reduced to ILS problem, thereby proving ILS is NP-hard. Thus proving the ILS problem to be NP-complete.

Solving the problem of target variables selection follows from solving the ILS problem. Thus, assuming that a graph exists for potential injection locations, the problem of selecting target variables is formally defined as an optimisation problem, TVS problem. The complexity of the verifying the correctness of the TVS problem has been proved to be $O(|U|^2)$, hence TVS is proven to be in NP. Next, the NP-hardness of the TVS problem is proved, by mapping and reducing the MDS problem to TVS problem. Thus, showing that the TVS problem is in NP and is NP-hard, and in turn proves the NP-completeness of the TVS problem.

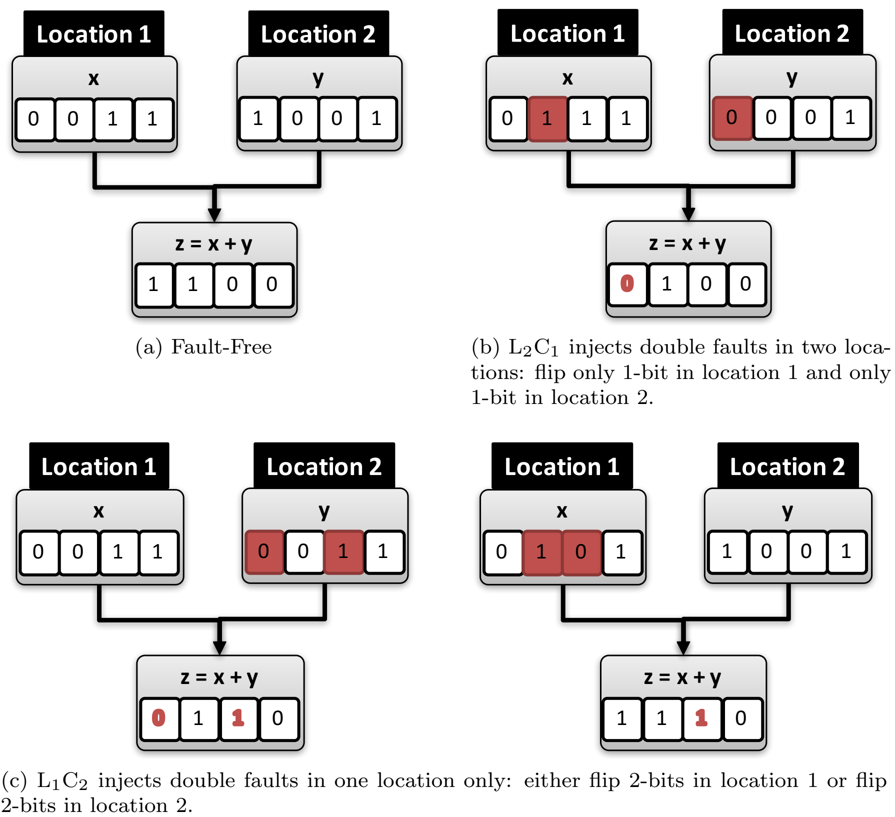
Following, this implies that exponential fault space for multiple soft-error injections can be made efficiently tractable in polynomial time. In subsequent chapters of this thesis, three approaches were developed to address the complexity formalised in this chapter: (i) In Chapter 6 the multiple soft-error injections fault space is made tractable by injecting only a maximum of two faults, the viability of the DBU faults for the $\mathrm{L}_n\mathrm{C}_m$ fault model was investigated, (ii) In Chapter 7, heuristics to solve the ILS problem and TVS problem for $\mathrm{L}_n\mathrm{C}_m$ fault model is developed, and (iii) In Chapter 8, the problem specification is weakened using data mining approach to further minimise the fault injection points.

CHAPTER 6

Double Single Bit-Flips (L_1C_2) Fault Model

The suppositions of the kind of faults a system is prone to and the way these assumed faults may influence the system is key to the design of fault tolerance mechanisms for the system. Fault tolerance mechanisms are evaluated with respect to the assumptions their design was based on. Should any assumptions on which a supposedly fault tolerant system design is based prove to be false, the system may likely fail to achieve its fault tolerance objectives. In the previous chapter, complexity analysis for selecting fault injection locations for multiple faults was done and it demonstrated that the fault space for multiple fault injections may be tractable. Thus, this chapter attempts to ascertain whether it is worth considering multiple faults for fault tolerance validation, i.e., to determine if the failure profile induced by the multiple fault model deviates from the failure profile single faults induce. To this end, this chapter proposes a double fault model for soft-errors that can be used for the design and validation of fault tolerance for embedded software systems. As mentioned, the main goal of this chapter is to consider whether the L_nC_m fault model may potentially be a viable fault model to be considered for software dependability validation. As such, the double faults error is considered because it is a tractable version of the L_nC_m fault model, and has been mentioned double faults error can occur either as double bit-flips in a single location, L_1C_2 or as two separate single bit-flips in two different locations, L_2C_1, (see Figure 6). Research has started looking at the former version of the double faults, thus the work presented in this chapter focused on the latter version in order to determine whether both

versions of double faults are the same (in terms of the failure profile they induce). Moreover, to keep the injection location space tractable, the chapter focuses on 3-*dimension* location space, i.e., early block, central block and late block. In order to demonstrate the viability of this fault model for fault tolerance validation, the fault model is introduced into seven embedded modules and analysed for error resilience with respect to failure mode. Analysis of the results presented indicates that the failure profile induced by the new fault model differs from the profile induced by existing fault models, indicating that the proposed fault model in this chapter is relevant during software dependability validation. Thus, the next chapter develops heuristics to address the problems associated with selecting efficient locations for multiple soft-errors injections presented in Chapter 5, by leveraging the information presented in this chapter.



(a) Fault-Free

(b) L_2C_1 injects double faults in two locations: flip only 1-bit in location 1 and only 1-bit in location 2.

(c) L_1C_2 injects double faults in one location only: either flip 2-bits in location 1 or flip 2-bits in location 2.

Figure 6.1: An overview of double faults.

6.1 Evaluation of Fault Models and Failure - Modes

As previously mentioned, fault injection is a widely used technique in dependability evaluation. Recent research have shown that multiple fault injections can be very effective in detecting software vulnerabilities [9, 163]. Other works have investigated impact of device-level fault injections that manifest as single bit-upsets in registers and main memory [30, 32, 139].

Current techniques in SWIFI-based FI for multiple bit-flips make use of simplifying decisions to keep the size of the fault space tractable. Recently, the effects of MBUs on SRAMs and DRAMs have been studied. In [78], the authors investigated DRAM disturbance errors that manifests as multiple bit-upsets in memory. On the other hand, the authors of [141] investigated the geometric effects of multiple bit-upsets injected into DRAMs. The main difference between the study in this thesis and these studies is the level of abstraction focused on. While these studies focus on fault effect on the circuit level, the work presented here is concerned with faults impacting the application level. The fault model under investigation in [141] is MBUs in multiple cells within the same memory location while that under investigation in [78] is MBUs in different memory locations. In spite of the fundamental differences between these works [78, 141] and the work presented here, they all showed higher rate of No Impact under the single bit-flip model. In addition, under the double bit-flip model, higher crash failure rate is observed. However, in these works [78, 141], it is reported that the proportion of SDCs is higher under the double bit-flip model, this is contrary to the findings of the work presented in this chapter. The work presented here reported a lower proportion of SDCs under the variant of double bit-flip model studied here than when compared with the single bit-flip model. Another difference between these works and the work presented here is, while in [141] and [78] the target location is the memory, the research presented here

target the ISA registers.

Similar to the work presented in this chapter, Ayatolahi et al. [9] mimicked bit-flips in registers of a real hardware platform. In addition, they investigated the impact of L_1C_1 and L_1C_2 on program execution. The work presented here differs from that presented in [9] mainly in the DBU fault model assumed. The DBU fault model in [9] selects a single location and flips two bits in that location, while in this work, in addition to the L_1C_1 model assumed in [9], the L_2C_1 fault model that chooses two locations and flips one bit in each location is also assumed. Another difference is: in [9], faults are also injected in memory words and the bit-error sensitivity for different target locations is investigated, in order to provide an insight regarding the results. Both the work presented here and that presented in [9] reported a higher level of benign (No Impact) executions for SBUs and a higher proportion of crash failures for DBUs.

6.2 Case Studies

As an assessment on how fault models impact on program executions, a series of experiments is conducted, using soft-error injections on seven embedded software systems where single bit and double bits errors were injected into CPU registers of the target systems. The aim of the study is to investigate how failure mode varies for different fault models. The first set of evaluations focused on how error resilience and error sensitivity varies for the different fault models, and the second set of evaluations focused how error error resilience and error sensitivity varies for different target locations. Error sensitivity is taken to be the probability that a fault causes software system failure as defined in Chapter 3 and error resilience taken to be the probability that a fault does not result in a program failure defined under the failure scheme in Chapter 3.4.3 .

The target programs used and fault models assumed are described in Chapter 3

(See Section 3.2). This section describes the parameters adopted for the target programs to use them with the fault injection tool described in Chapter 3 (See Section 3.4.2.

6.2.1 System Instrumentation

The LLFI tool was used for all fault injection experiments conducted in this thesis. The fault injection experiments reported in this chapter are undertaken on seven programs: derivatives, step, cubic, Isqrt, Corners, Step and Smoothing. To perform fault injection with LLFI, the source code of the software system is first compiled into a single IR byte-code. Twelve variables were instrumented in each target program. A golden run was created for each program. Three fault models, L_1C_1, L_1C_2 and L_2C_1 (See Chapter 3.2), were adopted for the fault injections experiments in this chapter. In line with the fault models assumed in this chapter, bit-flip faults were injected into bit-positions for all instrumented program variables. Nine different input sets are selected for each target program. The combination of input and target program is called an execution flow. This means, for each target program under each fault model experiments were conducted for nine execution flows. The target systems and their input data are described in Section 3.3.

For the L_1C_1 model, each fault injection input execution flow entailed a single bit-flip in a program variable at one bit-position, i.e., no multiple fault are introduced in any single input execution flow. For the L_1C_2 model, each fault injection input execution flow entailed a double bit-flip in a program variable at two bit-positions, i.e., not more than two faults are introduced in any single program execution, and only a single variable can be targeted. For the L_2C_1 model, each fault injection input execution flow entailed single bit-flip in two program variables at one bit-position each, i.e., exactly two variables are targeted in any single input execution flow, and no multiple faults can be inserted

in any variable.

6.2.2 Experimental Procedure

Before commencing the fault injection experiments, the CFG of IR byte-code of the program was partitioned into three parts, namely: (i) early, (ii) central and (iii) late. These partitions are defined as *block locations* (or blocks for short). From each block location, four target locations, were choosen at random, i.e., target locations are partitioned and selected according to their placement in the IR byte-code of the program, and also according to their execution location in the program. A *target location* (or location for short) is defined as a given register used by the program. When an L_1C_1 error is injected, a single location is selected. On the other hand, two locations are selected for L_2C_1 errors. A fault injection *experiment* is the injection of an error under the assumed fault model in a given target location or pair of locations. A fault injection *campaign* for a fault model is a set of experiments for a given input and program, i.e., a set of experiments for an execution flow.

Once a location (or pairs of locations) have been selected, bit-flip errors were then injected exhaustively in the locations to cover all possible combination. Errors are only injected in target location(s) immediately before the target location is read to avoid unnecessary overwrites. For each selected location, fault is injected only once during the execution of the program. For L_1C_1 and L_1C_2 errors, there are twelve target locations for each campaign and for L_2C_1 errors there are $\binom{x}{r}$ target locations pairs, $x = 12$ being the number of all chosen target locations and $r = 2$, the number of locations to target in any given experiment. Under the L_1C_1 fault model, n experiments were conducted in each target location, n being the length of the register. A total $48,384$ L_1C_1s were introduced in the various programs. For L_1C_2 errors, $\binom{n}{r}$ experiments were performed in each location, n being the size of the target location and $r = 2$ (the number of

bits to flip in the said location). A total of $1,524,096$ L_1C_2 errors were injected across the different target programs. Under the L_2C_1 model, for each location pair, $n \times m$ experiments were undertaken, m, n being the length of the target locations. Overall, a total of $17,031,168$ L_2C_1 errors were injected into the target programs.

Table 6.1: Register classificaiton scheme

Register	**Operation**	**Instruction Type**	**Data Type**
ADD/FADD	Returns the sum of its two operands.	Binary	Control or Other
SUB/FSUB	Returns the difference of its two operands.	Binary	Control or Other
MUL/FMUL	Returns the product of its two operands.	Binary	Control or Other
DIV/FDIV	Returns the quotient of its two operands.	Binary	Control or Other
SHL	Shifts a value to the left a specified number of its, and returns the shifted value.	Binary	Other
BITCAST	Converts a value to a second type without changing any bits. It returns the converted value.	Casting	Other
SITOFP	Converts a signed integer value to a floating point value. It returns the converted value.	Casting	Other
ZEXT	Zero extends its operand to a larger size, i.e., fills the higher order bits of the value with zero until it reaches the size of the destination type.	Casting	Other
LOAD	Specifies the memory address form which to load. The location of the memory pointed is loaded.	MAAC	Control or Other
ALLOCA	Allocates memory on the stack frame of the currently executing function and returns a pointer.	MAAC	Pointer
GETELEM	Gets the address of a subelement of an aggregate data structure. It performs address calculations only and does not access memory. It returns a pointer to an element.	MAAC	Pointer

Further, target locations were categorised based on register instruction type and the type of data held in the register. Table 6.1 depicts register classification scheme.

Register Instruction Type Categorisation

A target location is classified based on the type of operation it performs (see Table 6.1) as follows:

- **Binary**: If it performs binary or bitwise binary operations in a program, i.e., it performs computaions, such as adding, on two operands and returns the results of the operation.
- **Casting**: If it performs bit conversions operations, such as casting, converting value of one data type to another data type.
- **Memory Access and Address Computation (MAAC)**: If it loads data from memory, allocates memory on stack or gets address of a subelement of an aggregate data structure.

For L_2C_1 errors, target location pairs are classified as above only if both locations belong to the same category, and locations of mixed categories are classified as follows:

- **Binary and Casting**: If one location is Binary and the second is Casting.
- **Binary and MAAC**: If one target location is Binary and the other is MAAC.
- **Casting and MAAC**: If the target location is made of Casting and MAAC pair.

Register Data Type Categorisation

A target location is classified based on the type of data it holds (see Table 6.1) as follows:

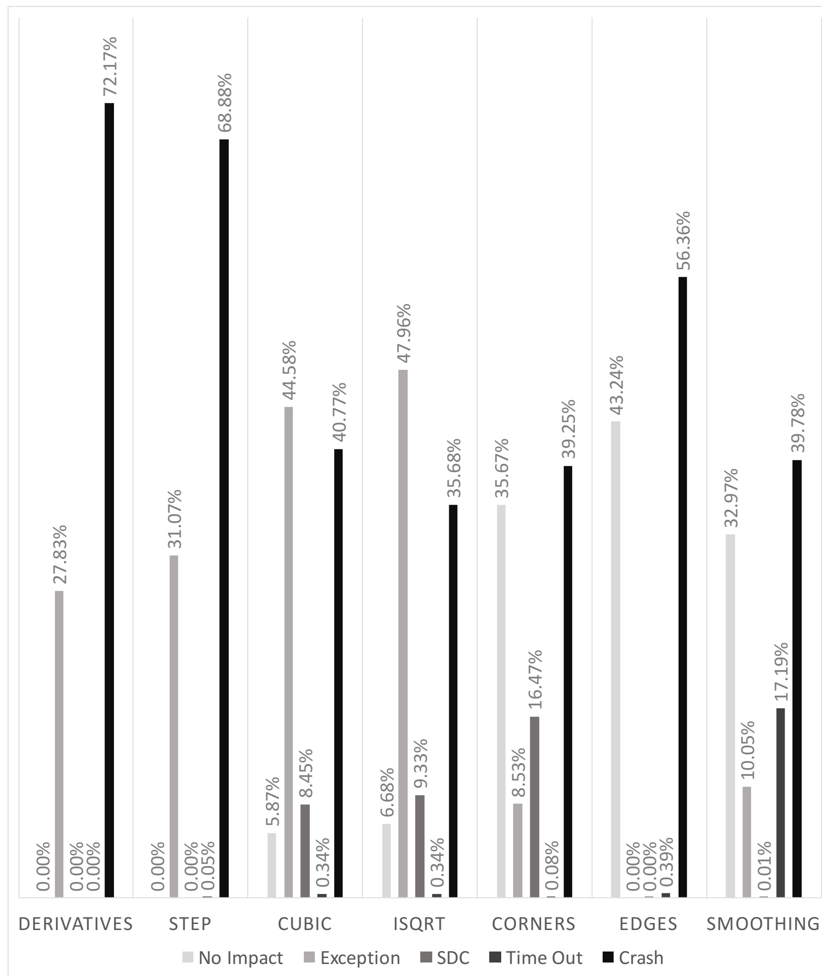
- **Pointer**: If it holds data that affects memory allocation or address computation. Pointer data include pointer, address computations for array and struct data items
- **Control**: If it holds data that would affect control flow. Control data items are usually loop termination condition and branching condition.
- **Other**: If it holds general value data that are neither pointer nor control. The data under this category are usually general value data such as signed and unsigned integer numbers etc.

For L_2C_1 errors, similar to instruction type categorisation, target location pairs are classified as any of the above only if both locations belong to the same category, and locations of mixed categories are classified as follows:

- **Pointer and Control**: If one target location holds Pointer data and the second holds Control data.
- **Pointer and Other**: If one location is Pointer and the other is Control.
- **Control and Other**: If the target location is made of Pointer and Control pair.

6.3 Impact of Fault Models

Three types of errors are injected into seven target programs. It should be mentioned that all target programs are without fault tolerance mechanisms. Further, a system without any specific fault tolerance implemented often exhibits a certain level of error tolerance (robustness) due to the fact that errors can be overwritten or that the system has a built-in resiliency against errors. Hence, experiments resulting as No impact may be due to this internal robustness of the program. The investigation shows that the different fault models do induce different failure profiles, it reveals that:



(a) L_1C_1

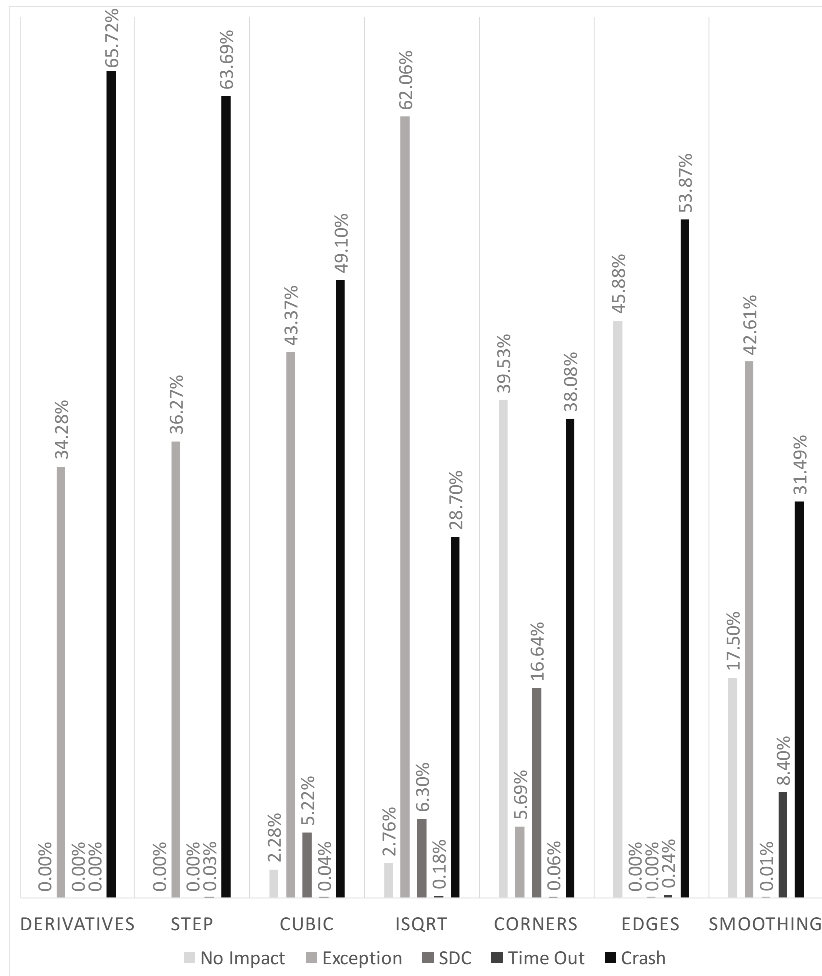
Figure 6.2: Error sensitivity distribution of the various fault models for each programs.

1. Double faults uncover more failures than the single faults,

2. L_1C_2 faults induce more SDCs than L_2C_1,

3. L_2C_1 faults cause more severe failures than L_1C_2.

Thus, motivating the need to adopt various fault models in software dependability validation and to extend these specific cases of the L_nC_m fault model. The remainder of this section presents and further discuses the results of impact of the different fault models investigated.

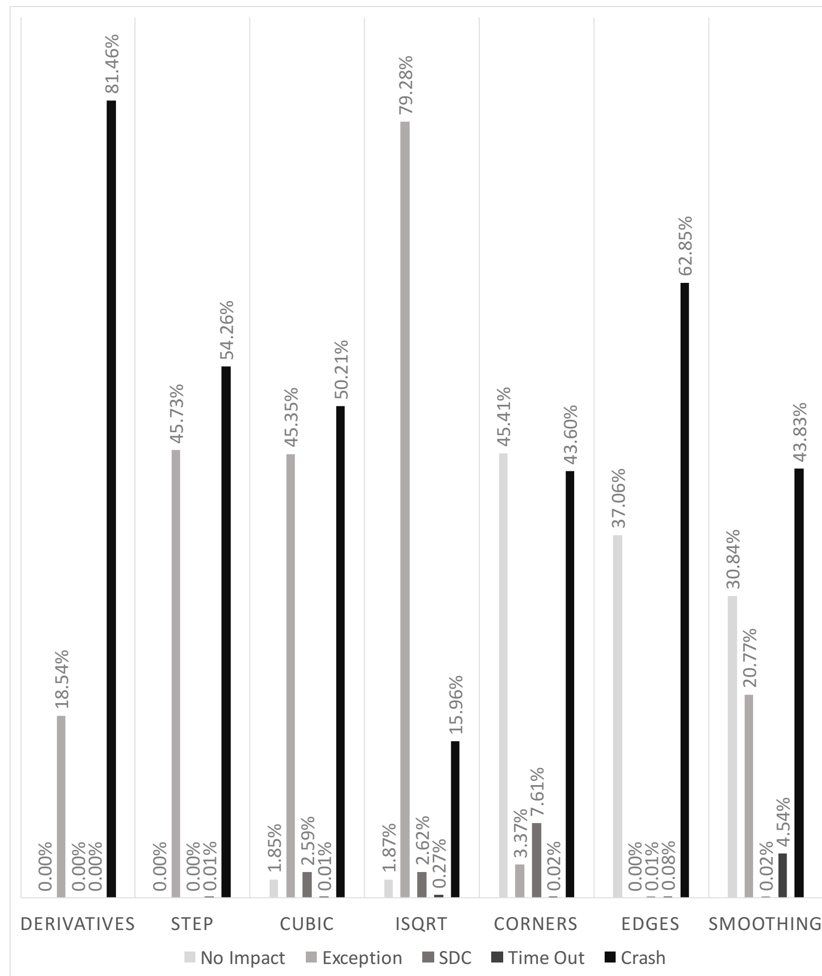


(b) L_1C_2

Figure 6.2: Error sensitivity distribution of the various fault models for each program.

Figures 6.2a–6.2c show the error sensitivity distribution for each fault model over the different programs. The vertical axis shows the percentage of experiments that fall in different failure classification for different target program represented in the horizontal axis.

Table 6.2 shows a summary of error resilience for the different failure classes under the different fault models. Each row shows the percentage of experiments that results in different error classifications for the different fault models. Table 6.3 shows the result for one way analysis of variance (ANOVA) tests per-



(c) L_2C_1

Figure 6.2: Error sensitivity distribution of the various fault models for each program.

formed to statistical test the effect of fault models on fault outcomes. Each row depicts the results of the significance of the effect of fault model on the fault outcome under the *Dependent Variable* column, it should be mentioned that the first row shows the results of fault model on no particular fault outcome. The p-value column determines whether the effect of the model is significant with 99% of confidence interval, the null hypothesis is accepted if the the p-value is greater than 0.01 and rejected otherwise. The F column depicts the Wilks' lambda value for the ANOVA test. The test are based on the linearly

independent pairwise comparisons among the estimated marginal means, the difference between these means is shown in Table 6.5. Negative mean difference denotes under the fault model *J*, the mean of the percentage of error resilience for the given error is higher than under the fault model *I*. The p-value column determines whether the mean difference is significant with 99% of confidence interval. The mean difference is significant when the p-value is less than 0.01 and insignificant otherwise. Table 6.4 shows the means, standard errors and confidence interval of experiments that fail in different error classifications over all programs for the fault models. Due to the large number of experiments (≈ 48000 for L_1C_1 errors, over 1.5 *million* for L_1C_2 errors and excess of 17 *million* for L_2C_1 errors), the 99% confidence interval for the measures in this section varies from $\pm 1.02\%$ to $\pm 4.49\%$ for L_1C_1 and L_1C_2 errors and from $\pm 0.65\%$ to $\pm 2.84\%$ for L_2C_1 errors.

6.3.1 L_2C_1 vs. L_1C_2 vs L_1C_1

Figures 6.2a, 6.2b and 6.2c present the overall error sensitivity for L_1C_1, L_1C_2 and L_2C_1 errors respectively. The error sensitivity for a fault outcome is calculated as shown in equation 6.1.

$$E_S = \frac{F_f}{F_N} \tag{6.1}$$

Here F_N is the total number of faults injected, i.e., total number of experiments, and F_f is the total number of experiments that results in the given fault outcome class.

The results from these figures show that observed failures are not uniform across the different programs. For example, errors injected into Derivatives consistently ended up as either Exception or Crash. Further, campaigns with Step never resulted in No impact or SDC. On another hand, the proportion of Time out

Table 6.2: Error resilience distribution of all fault models

Fault Model	Target Program	Fault Outcome (%)				
		No Impact	**Exception**	**SDC**	**Time Out**	**Crash**
L_1C_1	*Derivatives*	100.00	72.17	100.00	100.00	27.83
	Step	100.00	68.93	100.00	99.95	31.12
	Cubic	94.13	55.42	85.65	99.66	59.23
	Isqrt	93.32	52.04	85.42	99.66	64.32
	Corners	64.33	91.47	72.86	99.92	60.75
	Edges	56.76	100.00	99.99	99.61	43.64
	Smoothing	67.03	89.95	99.98	82.81	60.22
L_1C_2	*Derivatives*	100.00	65.72	100.00	100.00	34.28
	Step	100.00	63.73	100.00	99.97	36.31
	Cubic	97.72	56.63	89.74	99.96	50.90
	Isqrt	97.24	37.94	91.14	99.82	71.30
	Corners	60.47	94.31	73.09	99.94	61.92
	Edges	54.12	100.00	99.99	99.76	46.13
	Smoothing	82.50	57.39	99.99	91.60	68.51
L_2C_1	*Derivatives*	100.00	81.46	100.00	100.00	18.54
	Step	100.00	54.27	100.00	99.99	45.74
	Cubic	98.15	54.65	94.80	99.99	49.79
	Isqrt	98.13	20.72	96.87	99.73	84.04
	Corners	54.59	96.63	86.50	99.98	56.40
	Edges	62.94	100.00	99.98	99.92	37.15
	Smoothing	69.16	79.23	99.96	95.46	56.17

is predominantly higher when L_1C_1s are injected and fewest when L_2C_1s are injected. Similarly, the observed rate of No impact is often highest for L_1C_1 error experiments and lowest for L_2C_1 errors. The percentage of Crash was almost consistently highest for experiments under the L_2C_1 fault model. Similarly, frequency of Exception is almost regular highest when L_2C_1 errors are injected. Further, the results denote SDCs to results less from L_2C_1 errors.

Table 6.2 shows a summary of error resilience results under each fault model. The error resilience for a fault outcome is calculated as in equation 6.2.

$$E_R = 1 - E_S \tag{6.2}$$

If the error resilience of a program to a given fault outcome is considered to be distributed as a normal variable with a mean value equals to the quotient

Table 6.3: Null hypotheis test results for fault model effect on error resilience

Dependent Variable	F	P-value ($\alpha = 0.01$)	Result
	62.933	0.000	Reject
No Impact	0.648	0.523	Accept
Exception	10.128	0.000	Reject
SDC	15.059	0.000	Reject
Time Out	244.816	0.000	Reject
Crash	9.120	0.000	Reject

The F tests the effect of fault models on fault outcomes. This test is based on the linearly independent pairwise comparisons among the estimated marginal means (see Table 6.5).

between the number of experiments in the said fault outcome category and the total number of experiments, ANOVA can be performed. ANOVA is performed to test whether there are error resilience rate difference between the programs under the three fault models by testing the null hypothesis H_0 which states: "the fault outcome of an experiment does not depend on the fault model assumed for the experiment". The results of ANOVA in Table 6.3 allows the the rejection of H_0 with a confidence of 99%, this means that there is a statistically significant difference in error resilience based on an assumed fault model. However, the results also allows the acceptance of H_0 that states: "the probability of an experiment resulting in No impact is dependent on the assumed fault model".

Table 6.5 presents a pairwise comparisons of mean difference between error resilience of the programs to the different fault classes. Considering the first row, for example, the mean difference between error resilience to No Impact under L_1C_1 (fault model I) and L_1C_2 (fault model J) is -2.1352. This means on the average the there's a higher propbaibilty of programs being error resilient to No Impact for L_1C_1 errors than for L_1C_2 errors.

For example, the results show there is no significant mean difference in resilience to No impact between the three fault models (see Table 6.5). However, the

Table 6.4: Estimated marginal means for error resilience of all fault models

Dependent Variable / Fault Outcome	Fault Model	Mean (%)	Std. Error	Confidence Interval (99%) Lower Bound	Confidence Interval (99%) Upper Bound
No Impact	$L_1 C_1$	82.371	1.331	78.936	85.805
	$L_1 C_2$	84.506	1.331	81.071	87.941
	$L_2 C_1$	83.562	0.842	81.390	85.734
Exception	$L_1 C_1$	74.464	1.369	70.931	77.997
	$L_1 C_2$	67.944	1.369	64.411	71.476
	$L_2 C_1$	75.082	0.866	72.847	77.316
SDC	$L_1 C_1$	95.233	0.511	93.913	96.552
	$L_1 C_2$	95.975	0.511	94.656	97.295
	$L_2 C_1$	92.989	0.323	92.155	93.823
Time Out	$L_1 C_1$	97.376	0.396	96.355	98.397
	$L_1 C_2$	98.714	0.396	97.693	99.735
	$L_2 C_1$	89.787	0.250	89.142	90.433
Crash	$L_1 C_1$	50.557	1.738	46.071	55.042
	$L_1 C_2$	52.862	1.738	48.376	57.347
	$L_2 C_1$	58.581	1.099	55.744	61.418

results (see Table 6.2) shows that error resilience to No Impact is often lower for L_1C_1 errors than for either L_1C_2 or L_2C_1 errors. This implies L_1C_1 errors are more likely to end in No Impact than either L_1C_2 or L_2C_1 errors. On the other hand, the results indicate that all three error types have similar percentage of error resilience to Time Out failures. This implies that likelihood of an Experiment causing a Time out is low irrespective of the error type.

The results (see Table 6.5) also depict the mean error resilience to Exception is significantly lower for L_1C_2 errors than for both L_1C_1 and L_2C_1 errors, however they do not show significant difference between means under L_1C_1 and L_2C_1 fault models. The results further depict that for both L_1C_1 and L_1C_2 errors, average percentage of error resilience to Exceptions is significantly lower than for L_2C_1 errors (see Table 6.2). Thus, implying that the likelihood of an experiment causing an Exception is higher when injected with either L_1C_1 or L_1C_2 errors than when a L_2C_1 was injected . The results also show that across the programs,

Table 6.5: Pairwise comparisons between mean for all fault models

Dependent Variable (Fault Outcome)	Fault Model I	Fault Model J	Mean Difference (I-J)	Std. Error	P-value ($\alpha = 0.01$)	Confidence Interval (99%) Lower Bound	Confidence Interval (99%) Upper Bound
No Impact	L_1C_1	L_1C_2	-2.1352	1.88248	0.493	-7.6311	3.3606
		L_2C_1	-1.1914	1.57500	0.730	-5.7895	3.4068
	L_1C_2	L_1C_1	2.1352	1.88248	0.493	-3.3606	7.6311
		L_2C_1	0.9439	1.57500	0.821	-3.6543	5.5420
	L_2C_1	L_1C_1	1.1914	1.57500	0.730	-3.4068	5.7895
		L_1C_2	-0.9439	1.57500	0.821	-5.5420	3.6543
Exception	L_1C_1	L_1C_2	6.5205*	1.93617	0.002	0.8679	12.1731
		L_2C_1	-0.6175	1.61991	0.923	-5.3468	4.1118
	L_1C_2	L_1C_1	-6.5205*	1.93617	0.002	-12.1731	-0.8679
		L_2C_1	-7.1380*	1.61991	0.000	-11.8673	-2.4086
	L_2C_1	L_1C_1	0.6175	1.61991	0.923	-4.1118	5.3468
		L_1C_2	7.1380*	1.61991	0.000	2.4086	11.8673
SDC	L_1C_1	L_1C_2	-0.7429	0.72312	0.560	-2.8540	1.3683
		L_2C_1	2.2437*	0.60501	0.001	0.4774	4.0100
	L_1C_2	L_1C_1	0.7429	0.72312	0.560	-1.3683	2.8540
		L_2C_1	2.9865*	0.60501	0.000	1.2202	4.7528
	L_2C_1	L_1C_1	-2.2437*	0.60501	0.001	-4.0100	-0.4774
		L_1C_2	-2.9865*	0.60501	0.000	-4.7528	-1.2202
Time Out	L_1C_1	L_1C_2	-1.3379	0.55958	0.045	-2.9715	0.2958
		L_2C_1	7.5887*	0.46818	0.000	6.2219	8.9556
	L_1C_2	L_1C_1	1.3379	0.55958	0.045	-0.2958	2.9715
		L_2C_1	8.9266*	0.46818	0.000	7.5597	10.2934
	L_2C_1	L_1C_1	-7.5887*	0.46818	0.000	-8.9556	-6.2219
		L_1C_2	-8.9266*	0.46818	0.000	-10.2934	-7.5597
Crash	L_1C_1	L_1C_2	-2.3048	2.45845	0.617	-9.4821	4.8726
		L_2C_1	8.0240*	2.05688	0.000	-14.0290	-2.0189
	L_1C_2	L_1C_1	2.3048	2.45845	0.617	-4.8726	9.4821
		L_2C_1	5.7192	2.05688	0.015	-11.7242	0.2858
	L_2C_1	L_1C_1	-8.0240*	2.05688	0.000	2.0189	14.0290
		L_1C_2	-5.7192	2.05688	0.015	-0.2858	11.7242

Based on estimated marginal means. (see Table 6.4
The error term is Mean Square(Error) = 761.538.
* The mean difference is significant

error resilience to Exception is often lower under L_1C_2s than under L_1C_1s. This implies that L_1C_2 errors are more likely to cause Exceptions than either L_1C_1 errors or L_2C_1 errors.

Table 6.5 also shows that mean percentage of error resilience to SDC is significantly higher under the L_2C_1 fault model than under either L_1C_1 or L_1C_2 fault models, and no significant mean difference for L_1C_1 and L_1C_2 errors. Similarly, the results demonstrate that for majority of the programs percentage error resilience is higher under L_2C_1s than under both L_1C_1s and L_1C_2s (see Table 6.2). This means that the occurrence of SDCs are more probable under both L_1C_1 and L_1C_2 fault models than under L_2C_1 fault. However, for Edges and Smoothing the percentage of error resilience to SDCs is similar across the three fault models. On the hand, similar percentage of error resilience has been observed under L_1C_1s and L_1C_2s.

On another hand, the results have shown there is a significant overall lower mean error resilience to Crash when L_1C_1 or L_1C_2 errors are injected than for L_2C_1 errors (see Table 6.5). The results also show (see Table 6.2), percentage of error resilience to Crash is almost consistently lower under L_2C_1 than under either L_1C_2 or L_1C_1 across the different programs with the exception of Step and Isqrt, which shows the reverse. Hence, this means experiments subjected to L_2C_1 errors had higher probability of ending in Crash than those imparted with L_1C_1 and L_1C_2 errors.

The overall reduction of No impact and SDC under L_2C_1s may be due to L_2C_1 errors causing more severe failures resulting in the system exiting prematurely. Similarly, because L_2C_1 and L_1C_2 errors mostly causes the program to prematurely exit, little and no executions tend to hang.

6.4 Impact of Injection Location

This section investigates the effect of injection location on failure mode. Hence, the effect on block locations, instruction type and data type are analysed. The impact of blocks on failure groups is measured with respected to error sensitivity

rate as described in the preceding section. Percentage of error resilience is used to measure the effects of register instruction type and data type on experiments. The investigation demonstrates that injection location do affect the failure profile, this means, injection in certain locations are more efficient in uncovering vulnerabilities and are likely to cause failures. This motivates the need for a systematic approach to select injection locations for the L_nC_m fault model. The remainder of the section presents and further discuses these results.

Table 6.7, Table 6.8 and Table 6.9 present the percentage error sensitivity observed under the L_1C_1, L_1C_2 and L_2C_1 fault models respectively. Each row represents the percentage of error sensitivity for the different failure classes under the fault outcome column for a given block location and target program. Table 6.6 depicts the confidence interval at 99% for the observed error sensitivity.

Figures 6.3a–6.3c depict effects of instruction type on the error sensitivity rate to the different failure classes with respect to the assumed fault model. The vertical axes represent the percentage of error sensitivity plotted against the failure classes.

Figures 6.4a–6.4c present the results of the effect of error injections into the different register data types on the different failure classes. Similar to the instruction type charts, the horizontal axes depicts percentage error sensitivity over the different failure classes represented on the vertical axes.

6.4.1 Block Location

This section evaluates the impact of faults with respect to their block location in the program execution. Tables 6.7–6.9 show the percentage of error sensitivity observed for the different fault models over the different target programs and Table 6.6 shows confidence interval ranging from ±3.34% to ±6.44% for L_1C_1

and L_1C_2 errors and from ±0.45% to ±9.10% L_2C_1 errors.

Table 6.6: Confidence interval of error resilience for all blocks

Fault Model	Block Location	Confidence Interval (99%)				
		No Impact	Exception	SDC	Time Out	Crash
L_1C_1	*Early* *Central* *Late*	±3.34%	±5.62%	±1.76%	±0.63%	±6.44%
L_1C_2	*Early* *Central* *Late*	±3.34%	±5.62%	±1.76%	±0.63%	±6.44%
L_2C_1	*Early* *Central*	±4.72%	±5.62% ±7.95%	±2.49%	±0.90%	±9.10%
	Late *Early + Central*	±2.36%	±3.97%	±1.25%	±0.45%	±4.55%
	Early + Late		±3.98%	±1.24%		
	Central + Late		±7.95%	±0.63%	±0.90%	±9.10%

Table 6.7: Error sensitivity distribution for different block locations under L_1C_1.

Target Program	Block Location	Fault Outcome (%)				
		No Impact	Exception	SDC	Time Out	Crash
Derivatives	*Early*	0.00	25.30	0.00	0.00	74.70
	Central	0.00	28.88	0.00	0.00	71.13
	Late	0.00	29.33	0.00	0.00	70.68
Step	*Early*	0.00	12.27	0.00	0.05	87.68
	Central	0.00	54.15	0.00	0.05	45.80
	Late	0.00	26.80	0.00	0.05	73.15
Cubic	*Early*	5.50	43.00	5.95	0.43	45.13
	Central	4.60	47.40	10.69	0.30	37.01
	Late	7.50	43.33	8.70	0.30	40.18
Isqrt	*Early*	7.94	16.43	8.59	0.43	66.61
	Central	4.60	65.77	10.69	0.05	18.89
	Late	7.50	61.70	8.70	0.55	21.55
Corners	*Early*	28.38	7.40	16.53	0.25	47.45
	Central	68.08	10.85	3.78	0.00	17.30
	Late	10.55	7.35	29.10	0.00	53.00
Edges	*Early*	8.72	0.00	0.01	1.17	90.10
	Central	54.22	0.00	0.00	0.00	45.78
	Late	66.79	0.00	0.00	0.00	33.20
Smoothing	*Early*	5.58	30.16	0.00	2.35	61.91
	Central	75.83	0.00	0.01	0.00	24.17
	Late	17.50	0.00	0.02	49.22	33.26

The results show that block location exhibits both similar and contrasting behaviours for the three types of errors. For example, it is observed that overall,

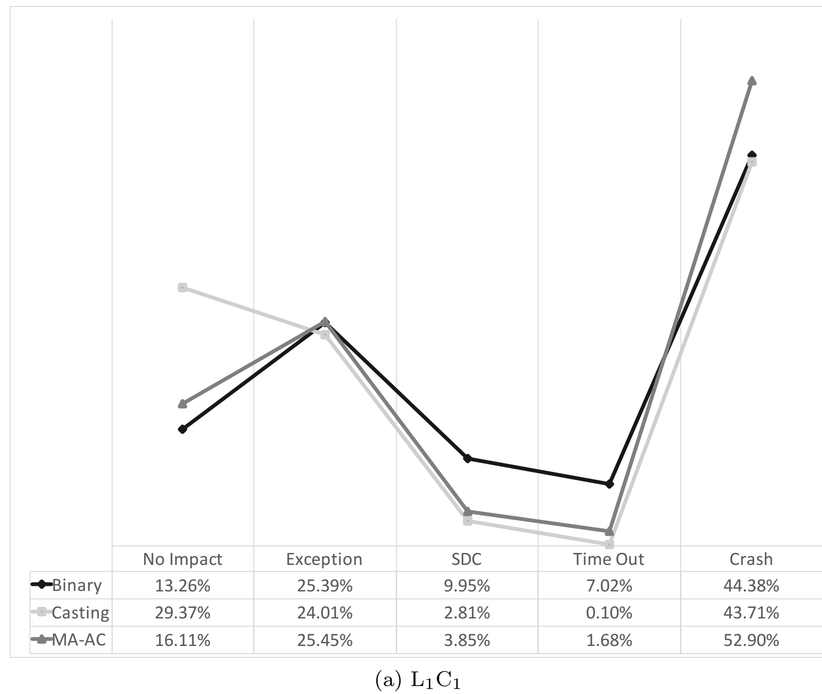
Table 6.8: Error sensitivity distribution for different block locations under L_1C_2.

Target Program	**Block Location**	**Fault Outcome (%)**				
		No Impact	**Exception**	**SDC**	**Time Out**	**Crash**
	Early	0.00	33.53	0.00	0.00	66.47
Derivatives	*Central*	0.00	34.75	0.00	0.00	65.25
	Late	0.00	34.55	0.00	0.00	65.45
	Early	0.00	4.85	0.00	0.03	95.12
Step	*Central*	0.00	69.43	0.00	0.03	30.53
	Late	0.00	34.53	0.00	0.03	65.43
	Early	2.39	41.90	3.18	0.11	52.42
Cubic	*Central*	2.54	44.16	6.37	0.00	46.93
	Late	1.91	44.04	6.10	0.00	47.95
	Early	2.90	39.20	3.87	0.13	53.90
Isqrt	*Central*	3.07	73.99	7.70	0.00	15.24
	Late	2.30	72.99	7.33	0.42	16.96
	Early	33.67	4.93	12.10	0.17	49.13
Corners	*Central*	75.47	7.23	3.60	0.00	13.70
	Late	9.45	4.90	34.23	0.00	51.42
	Early	8.70	0.00	0.01	0.73	90.56
Edges	*Central*	51.04	0.00	0.00	0.00	48.96
	Late	77.91	0.00	0.00	0.00	22.09
	Early	1.84	37.32	0.00	1.47	59.38
Smoothing	*Central*	41.01	43.98	0.01	0.00	15.00
	Late	9.64	46.53	0.01	23.72	20.09

irrespective of the error type, early injections terminates with a Crash, and late injections results in higher proportion of Time outs. Further, the results also denote that failure mode for individual target programs tend to vary under all three fault models. This may be possibly be on the account of early injections increasing the likelihood of error propagation. Additionally, the results show an almost consistent concomitant increase in proportion of Crash with reduction of Exception rate and vice versa across the block locations. On the other hand, the results depict late injections of L_1C_1 and L_1C_2 errors induces higher percentage of SDCs whereas the opposite is the case for late injection of L_2C_1 errors. L_2C_1 error can be injected at different times. For, L_2C_1 errors injection in Central blocks rarely causes Crash, whereas injecting a L_2C_1 error in target combination of central and block of another location type greatly increase the proportion of Crash.

Table 6.9: Error sensitivity distribution for different block locations under L_2C_1.

Target Program	Block Location	Fault Outcome (%)				
		No Impact	Exception	SDC	Time Out	Crash
Derivatives	*Early*	0.00	0.15	0.00	0.00	99.85
	Central	0.00	13.63	0.00	0.00	86.38
	Late	0.00	13.35	0.00	0.00	86.65
	Early & Central	0.00	21.38	0.00	0.00	78.63
	Early & Late	0.00	22.04	0.00	0.00	77.96
	Central & Late	0.00	40.73	0.00	0.00	59.28
Step	*Early*	0.00	6.82	0.00	0.02	93.15
	Central	0.00	77.08	0.00	0.03	22.90
	Late	0.00	76.48	0.00	0.00	23.53
	Early & Central	0.00	11.76	0.00	0.02	88.22
	Early & Late	0.00	25.78	0.00	0.00	74.22
	Central & Late	0.00	76.48	0.00	0.00	23.53
Cubic	*Early*	1.42	44.64	1.54	0.02	52.38
	Central	2.01	43.52	2.95	0.00	51.51
	Late	2.03	48.64	3.17	0.00	46.16
	Early & Central	1.74	44.64	2.38	0.01	51.23
	Early & Late	1.73	43.52	2.25	0.01	52.49
	Central & Late	2.14	47.15	3.24	0.00	47.48
Isqrt	*Early*	1.48	79.02	1.60	0.02	17.88
	Central	2.06	81.28	3.02	0.00	13.64
	Late	2.06	81.92	3.22	0.30	12.50
	Early & Central	1.79	81.36	2.44	0.01	14.40
	Early & Late	1.77	70.15	2.31	0.49	25.29
	Central & Late	2.08	81.92	3.16	0.78	12.05
Corners	*Early*	15.80	2.96	10.56	0.10	70.58
	Central	81.88	4.34	0.76	0.00	13.02
	Late	6.73	4.46	0.76	0.00	88.05
	Early & Central	44.64	2.00	8.86	0.00	44.51
	Early & Late	39.79	1.98	23.82	0.00	34.42
	Central & Late	83.63	4.46	0.91	0.00	10.99
Edges	*Early*	0.00	0.00	0.00	0.37	99.63
	Central	16.46	0.00	0.00	0.00	83.54
	Late	88.17	0.00	0.00	0.00	11.83
	Early & Central	8.52	0.00	0.01	0.10	91.36
	Early & Late	40.47	0.00	0.02	0.00	59.52
	Central & Late	68.75	0.00	0.00	0.00	31.25
Smoothing	*Early*	0.00	24.83	0.00	0.85	74.33
	Central	89.71	0.00	0.01	1.08	9.20
	Late	17.74	0.00	0.08	14.69	67.49
	Early & Central	7.32	43.94	0.00	0.31	48.43
	Early & Late	0.96	55.85	0.01	4.69	38.50
	Central & Late	69.31	0.00	0.02	5.63	25.04



	No Impact	Exception	SDC	Time Out	Crash
Binary	13.26%	25.39%	9.95%	7.02%	44.38%
Casting	29.37%	24.01%	2.81%	0.10%	43.71%
MA-AC	16.11%	25.45%	3.85%	1.68%	52.90%

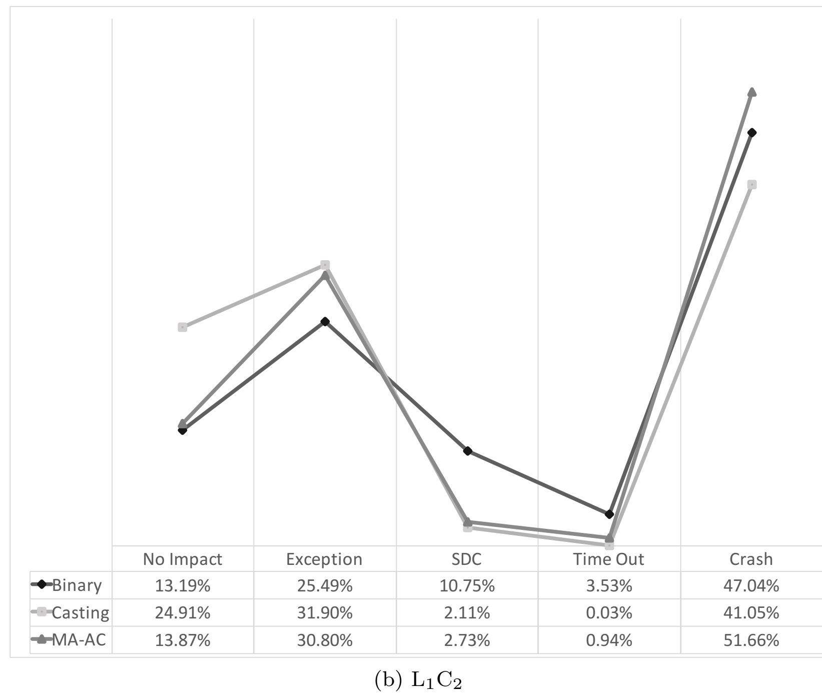
(a) L_1C_1

Figure 6.3: Error sensitivity distribution of instruction type for the fault models over all target programs.

6.4.2 Register Instruction Type

Figure 6.3a presents the impact of instruction type of the target location when imparted with L_1C_1 errors. The results, for example, implies that Crash rate is higher in MAAC operations. It may be argued that errors injected in these type of locations are more severe because these instructions are most probably pointers to stack and other register addresses. On the hand, the results show that SDC rate are most probably when injected in Binary locations. This may be on the account of Binary targets are usually value which may likely affect the output program. In Figure 6.3b, the result showing how injecting L_1C_2 errors into the different instruction type affect the failure classes is presented. The results show that subjecting the different instruction types to L_1C_2 errors causes similar failure mode.

Figure 6.3c depicts the failure mode when the different instruction type are tar-



	No Impact	Exception	SDC	Time Out	Crash
Binary	13.19%	25.49%	10.75%	3.53%	47.04%
Casting	24.91%	31.90%	2.11%	0.03%	41.05%
MA-AC	13.87%	30.80%	2.73%	0.94%	51.66%

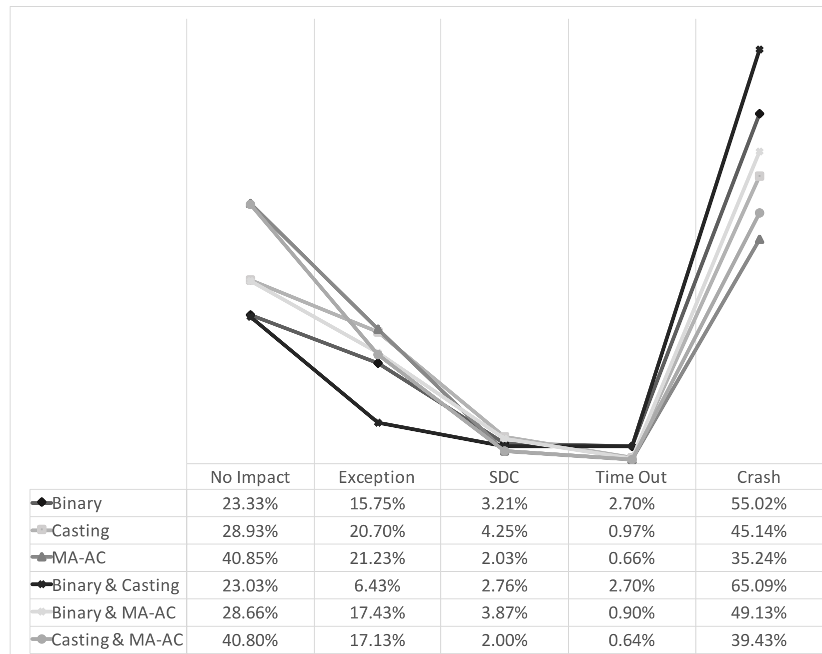
(b) L_1C_2

Figure 6.3: Error sensitivity distribution of instruction type for the fault models over all target programs.

geted with L_2C_1 errors. Similar to block locations, L_2C_1 errors have instruction type combination unique to them, because, they can have one target in an instruction type of one category and the second target in instruction type of a different category. The results show, dissimilar to L_1C_1 and L_1C_2 errors, Crash rate is higher when injection is in a combination of Binary and Casting location.

6.4.3 Register Data Type

The results in Figure 6.4a depict the effect of L_1C_1 error injections into the different data types on the different failure classes. The results, for example, show that L_1C_1 errors in Pointer often result in Crash, L_1C_1 errors in Control cause higher proportion of No impact and in Other induce more SDC rate. Arguably, L_1C_1 errors in pointers tend to cause more severe failures. Figure 6.4b demonstrates that the effect of the different data types on the fault outcome



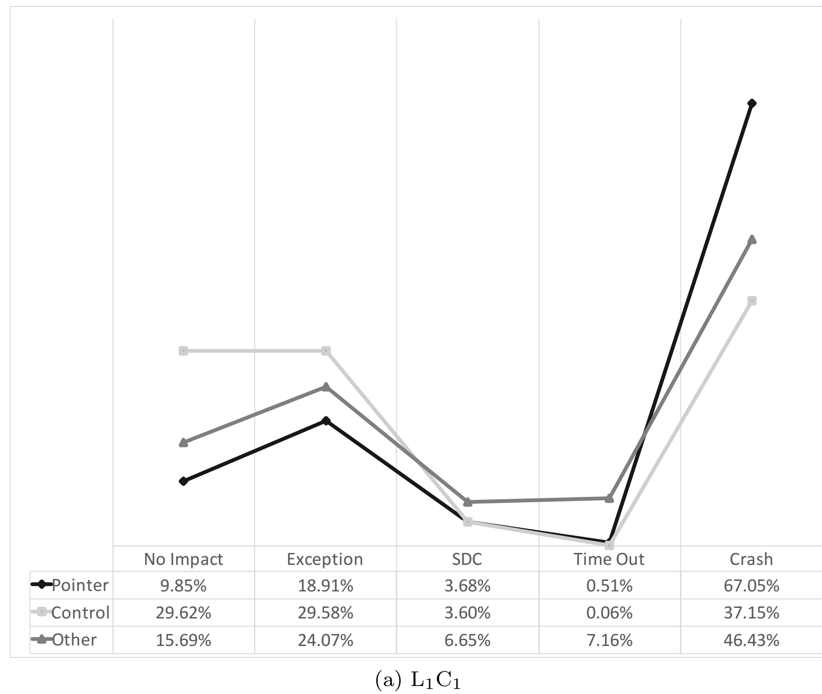
	No Impact	Exception	SDC	Time Out	Crash
Binary	23.33%	15.75%	3.21%	2.70%	55.02%
Casting	28.93%	20.70%	4.25%	0.97%	45.14%
MA-AC	40.85%	21.23%	2.03%	0.66%	35.24%
Binary & Casting	23.03%	6.43%	2.76%	2.70%	65.09%
Binary & MA-AC	28.66%	17.43%	3.87%	0.90%	49.13%
Casting & MA-AC	40.80%	17.13%	2.00%	0.64%	39.43%

(c) L_2C_1

Figure 6.3: Error sensitivity distribution of instruction type for the fault models over all target programs.

when imparted with L_1C_2 errors is similar to the effect caused by injecting L_1C_1 errors.

Figure 6.4c show how injecting L_2C_1 errors into the various data type impact on failure mode. The results show that, similar to L_1C_2 and L_1C_1 errors, L_2C_1s cause higher Exception rate when injected into Other. The results demonstrate, dissimilar to L_1C_2 and L_1C_1 errors, L_2C_1s cause higher No impact rate when injected into Other or when inserted into combination of Control and Other. The results also show that although injecting L_2C_1 in Pointer cause high proportion of Crash, injecting L_2C_1 errors into a Pointer and Control target combination induces higher proportion of Crash.



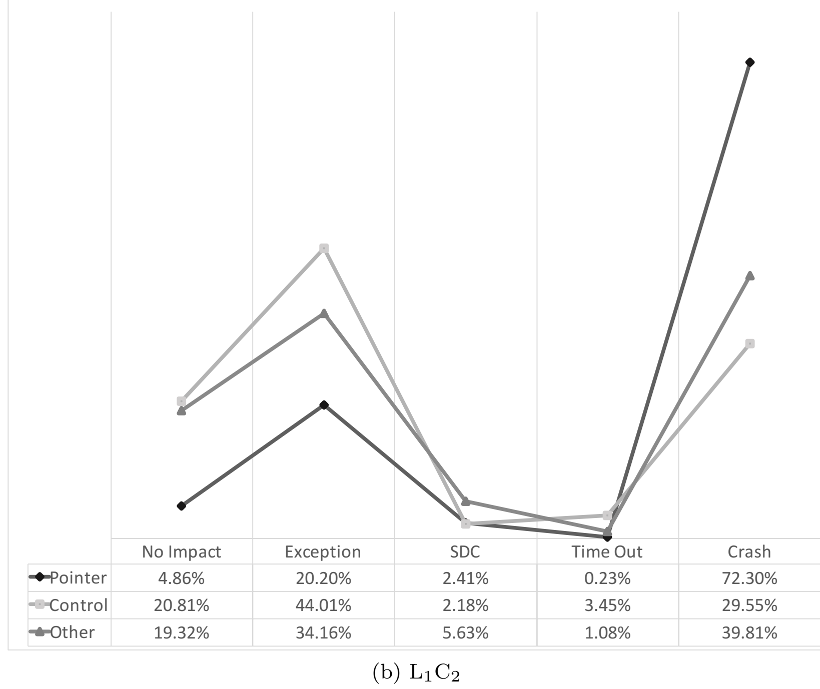
	No Impact	Exception	SDC	Time Out	Crash
Pointer	9.85%	18.91%	3.68%	0.51%	67.05%
Control	29.62%	29.58%	3.60%	0.06%	37.15%
Other	15.69%	24.07%	6.65%	7.16%	46.43%

(a) L_1C_1

Figure 6.4: Error sensitivity distribution of data type for the fault models over all target programs.

6.5 Correlations

In this section two sets of correlations analysis was done to test: (i) the linear relationship between failure classifications, and (ii) testing the monotonic relationship between injection locations with respect to error sensitivity rate and failure classes. Correlation coefficient is used to measure the correlation relationships, i.e., correlation coefficient is used as a measure to test the extent to which two variables tend to change together. A correlation coefficient describes both the strength and the direction of the association. *Pearson correlation coefficient* is adopted to measure linearity between failure classes, and *Spearman's rank-order correlation coefficient* to test association between injection locations with respect to failure classes and their rate. To this the percentage of error sensitivity is ranked starting from 1 being the highest. when more than one sample share the same error sensitivity percentage, they are then given the same rank.



	No Impact	Exception	SDC	Time Out	Crash
Pointer	4.86%	20.20%	2.41%	0.23%	72.30%
Control	20.81%	44.01%	2.18%	3.45%	29.55%
Other	19.32%	34.16%	5.63%	1.08%	39.81%

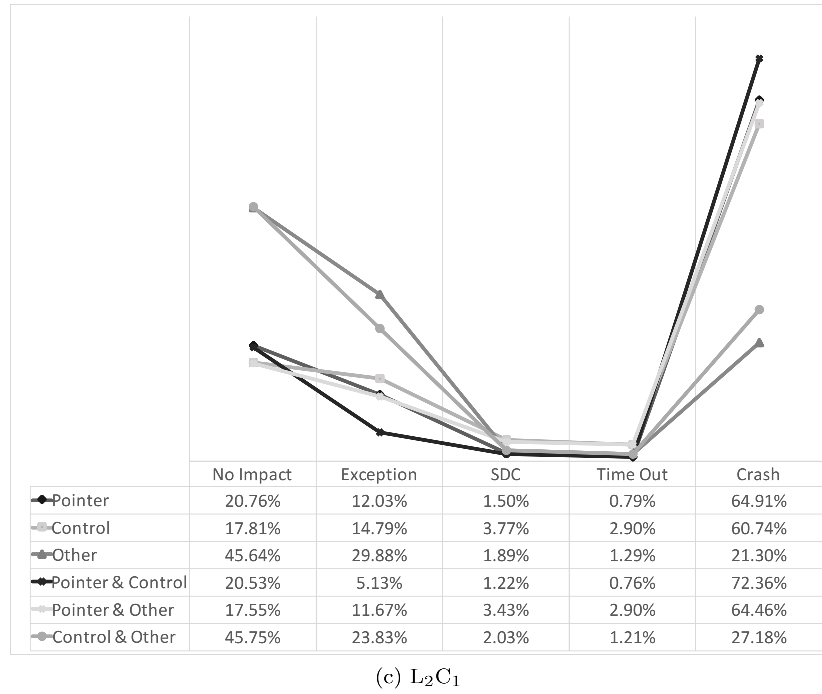
(b) L_1C_2

Figure 6.4: Error sensitivity distribution of data type for the fault models over all target programs.

Spearman's rank-order correlation coefficient: The Spearman's correlation evaluates the monotonic relationship between two continuous or ordinal variables. In a monotonic relationship, the variables tend to change together, but not necessarily at a constant rate. There are two methods to calculate Spearman's rank-order correlation depending on whether: (i) the data does not have tied ranks or (ii) the data has tied ranks. The formula in equation 6.3 measures the ρ when there are no tied ranks:

$$\rho = 1 - \frac{6 \sum d_i}{n(n^2 - 1)} \tag{6.3}$$

where d_i = difference in paired ranks and n = number of cases. Equation 6.4 shows the formula to calculate ρ when there are tied ranks:



	No Impact	Exception	SDC	Time Out	Crash
Pointer	20.76%	12.03%	1.50%	0.79%	64.91%
Control	17.81%	14.79%	3.77%	2.90%	60.74%
Other	45.64%	29.88%	1.89%	1.29%	21.30%
Pointer & Control	20.53%	5.13%	1.22%	0.76%	72.36%
Pointer & Other	17.55%	11.67%	3.43%	2.90%	64.46%
Control & Other	45.75%	23.83%	2.03%	1.21%	27.18%

(c) L_2C_1

Figure 6.4: Error sensitivity distribution of data type for the fault models over all target programs.

$$\rho = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2(y_i - \bar{y})^2}} \tag{6.4}$$

where i = paired score. The ρ is based on the ranked values for each variable rather than the raw data. A ρ of $+1$ indicates a perfect association of ranks, a ρ of zero indicates no association between ranks and a ρ of -1 indicates a perfect negative association of ranks. The stronger the association of the two ranks, the closer ρ will be to either $+1$ or -1 depending on whether the relationship is positive or negative, respectively.

Table 6.10 shows the ρ and how significant ($p-value$) is the association between ranks. ρ is the correlation between the two variables (one listed in the row, the other in the column). ρ marked with $**$ means the association is significant at $\alpha = 0.01$.

Table 6.10: Spearman's rank-order correlations

		Instruction Type	Data Type	Block Location	Fault Model
Instruction Type	ρ	1.000	0.270**	0.332**	0.498**
	p-value	.	0.000	0.000	0.000
Data Type	ρ	0.270**	1.000	0.462**	0.576**
	p-value	0.000	.	0.000	0.000
Block Location	ρ	0.332**	0.462**	1.000	0.619**
	p-value	0.000	0.000	.	0.000
Fault Model	ρ	0.498**	0.576**	0.619**	1.000
	p-value	0.000	0.000	0.000	.

** Correlation is significant at the 0.01 level (2-tailed).

Pearson product moment correlation: The Pearson correlation evaluates the linear relationship between two continuous variables, X, Y. A correlation is linear when a change in one variable is associated with a proportional change in the other variable. The Pearson correlation coefficient can take a value between $+1$ and -1 inclusive, where $+1$ is total positive correlation, zero is no correlation, and -1 is total negative correlation. A value greater than zero indicates a positive relationship; i.e., as the value of one variable increases, so does the value of the other variable. A value less than 0 indicates a negative association, i.e., as the value of one variable increases, the value of the other variable decreases. Pearson's correlation coefficient when applied to a sample is commonly represented by the letter r. Equation 6.5 shows how r is measured:

$$r = \frac{n\sum x_i y_i - \bar{x}\bar{y}}{\sqrt{(\sum {x_i}^2 - n\bar{x}^2)(\sum {y_i}^2 - n\bar{y}^2)}} \tag{6.5}$$

where n = number of cases. Similar to the Spearman's correlation coefficient, the closer r is to zero, the weaker the association between the variables.

Table 6.11 depicts the r and the significance ($p - value$) of the association between variables. r the linear correlation between the two variables (one listed in the row, the other in the column). Similar with the Spearman's rank-order,

Table 6.11: Pearson product moment correlations

		No Impact	Exception	SDC	Time Out	Crash
No Impact	r	1.000	-0.373**	-0.133**	-0.162**	-0.328**
	p-value		0.000	0.000	0.000	0.000
Exception	r	-0.373**	1.000	-0.183**	-0.254**	-0.426**
	p-value	0.000		0.000	0.000	0.000
SDC	r	-0.133**	-0.183**	1.000	0.530**	-0.371**
	p-value	0.000	0.000		0.000	0.000
Time Out	r	-0.162**	-0.254**	0.530**	1.000	-0.368**
	p-value	0.000	0.000	0.000		0.000
Crash	r	-0.328**	-0.426**	-.0371**	-0.368**	1.000
	p-value	0.000	0.000	0.000	0.000	

** Correlation is,significant at the 0.01 level (2-tailed).

$p - value$ of r marked with $**$ means the association is significant at $\alpha = 0.01$.

6.5.1 Testing Monotonic Relationships

The Spearman's rank-order correlation coeffient was used to measure the association between injection locations and models (see Table 6.10). It should be mentioned that statistical significance does not indicate the strength of the Spearman's rank-order correlation. In fact, the statistical significance testing of the Spearman's correlation does not provide you with any information about the strength of the relationship. Thus, achieving a value of $p = 0.001$, for example, does not mean that the relationship is stronger than if a value of $p = 0.04$ is achieved. This is because the significance test is investigating whether one can reject or fail to reject the null hypothesis, H_0. For example, setting $\alpha = 0.05$, achieving a statistically significant Spearman's rank-order correlation means that one can be sure that there is less than a 5% chance that the strength of the relationship found (ρ) happened by chance if the H_0 are true. Also note that, by definition, any variable correlated with itself has a correlation of 1.

Following are the ${H_0}^i$ tested for the correlation relationships:

- $\mathbf{H_0}^{\mathbf{1}}$: There is no relationship between fault model and block location

with respect to failure classes and their error sensitivity rate.

- $\mathbf{H_0}^{\mathbf{2}}$: There is no relationship between fault model and instruction type with respect to failure classes and their error sensitivity rate.
- $\mathbf{H_0}^{\mathbf{3}}$: There is no relationship between fault model and data type with respect to failure classes and their error sensitivity rate.
- $\mathbf{H_0}^{\mathbf{4}}$: There is no relationship between block location and instruction type with respect to failure classes and their error sensitivity rate.
- $\mathbf{H_0}^{\mathbf{5}}$: There is no relationship between block location and instruction with respect to failure classes and their error sensitivity rate.
- $\mathbf{H_0}^{\mathbf{6}}$: There is no relationship between instruction type and data type with respect to failure classes and their error sensitivity rate.

Here, strength of relationship is considered to be: (i) positively small or negatively small, if the value of ρ falls between 0.1 to 0.3 or between -0.1 to -0.3 respectively; (ii) positively medium or negatively medium, if ρ measures between 0.3 to 0.5 or -0.3 to -0.5 respectively, and (iii) positively large or negatively large, if ρ ranges between 0.5 to 1 or -0.5 to -1 respectively.

Table 6.10 denotes that ${H_0}^1$ through ${H_0}^6$ are rejected and the results show that the strength of correlation varies from small to large. For example, the results show that there is small positive association between instruction type and data type ($\rho = 0.277$), a medium positive relation between data type and block location ($\rho = 0.462$) and a large positive correlation between block location and fault model ($\rho = 0.619$). This implies that error sensitivity rate for failure classes is effected by (i) relationship of the instruction and data type of target locations, (ii) relationship of the data type and block location of target locations and (iii) relationship of the block location and fault model of target locations respectively.

6.5.2 Testing Linear Relationships

Pearson product moment correlation is used to measure the linear correlation between failure groups (see Table 6.11). Similar with the Spearman's rank-order, the significance level does not show indicate strength of the linear association is between the two variables. Further, the strength of relationship for r is classified similarly to how the strength of association for ρ is defined. However, for r, only associations of medium and large strength are considered. A variable correlated with itself will always have a correlation coefficient of 1. The null hypotheses are tested, one for each pairwise comparisons between failure classifications. For example, a null hypothesis to test the relationship between SDC and No impact states: "There is no linear relationship between error sensitivity rate of SDC and error sensitivity of No impact".

Table 6.11 shows that often the failure classes have statistically significant negative linear relationships with 99% confidence interval, which is expected. Thus, the analysis set criterion for acceptable correlation as $r > 0.3$ or $r > -0.3$ for positive and negative correlations respectively. The results show Crash to predominantly have statistically significant negative linear correlation with all other failure classes. For example, Crash strongest linear association is with Exception, $r = -0.426, p - value < 0.0001$. This means, overall Crash rate increases when Exception rate decreases, and vice versa.

Based on the correlation criterion adopted, the result depicts few linear associations between the other class of failures. For example, the results show a medium negative linear correlation between Exception and No impact, $r = -0.373, p - value < 0.0001$. The results also denote a positive linear association based on the analysis criterion set. For example, a large positive linear correlation between SDC rate and Time out, $r = 0.530, p - value < 0.0001$ was observed. This implies that SDC rate increases when Time out increases.

6.6 Implication and Limitation

The case studies presented have demonstrated that the fault model assumed have an impact on program execution. However, it should be noted that the results presented in this chapter are specific for the target programs, the fault models and input sets selected. For other programs, fault models, and/or input sets the results may vary. A second limitation in the results presented here is that, to the best of our knowledge, there is scarcity of field data that shows how multiple bit upsets will manifest themselves in registers. There is however increasing evidence that the rate of hardware errors is increasing. The relevance of the results presented in this chapter is only relevant as far as the field data matches the pattern of errors injection used. Another implication of the results is that it does not matter for the SDC rate of the program whether L_1C_1 or L_1C_2 faults are injected. This implies that if the primary focus is on SDCs, then L_1C_1 fault models may be sufficient for analysing the resilience of the program, compared to the double fault models.

6.7 Summary and Conclusions

In this chapter the failure modes for soft-error models is investigated. Two variants of double bit-flip errors have appeared in dependability evaluation recently. The objective, in this chapter, is to determine whether there is relevance in using both variants during validation. A range of errors injected into register are considered and the impact of the type of locations on failures investigated. The results presented in this chapter are statistically significant, especially after conducting ≈ 10.5 million fault injection experiments on seven embedded software. The analyses shows both variants are relevant as they induce different failure profile in software and, in some cases, the failures are unique to a given variant. Thus, the Double Single Bit-Flip (L_2C_1) fault model is proposed as novel fault

model for soft-error dependability evaluation and as a specific case of the L_nC_m fault model.

The results presented in this chapter indicate certain instruction type and certain data type (held in the register) are more resilient to certain errors than others. Similarly, certain block locations are more sensitive to errors than others. Further, fault models to be considered for validation, may be influenced by the failure class under focus.

To this point it has been shown that the injection location have an impact on failure profile of a software, and will provide insight into selecting potential target locations under a multiple locations multiple corruptions fault model. The next chapter develops an approach, using the information obtained in this chapter, to address the complexity issues associated with selecting potential locations and target variables for multiple fault injections, identified in Chapter 5.

CHAPTER 7

Towards Efficient Multiple Soft-Errors Injection

In this chapter, the problem of injecting multiple faults during an execution of a software system is addressed. Traditionally, for a single bit-flip, a location in the software is selected and a variable at that location is targeted for fault injection. During execution, when the location is reached, a fault is injected into that target variable.

On the other hand, when multiple fault injection is considered, there is both a spatial and temporal dimension that have to be taken into account [163]. In general, it means that more than one variable can be targeted at a given location (spatial dimension) or fault injection can happen in several locations at different times (temporal dimension). Illustrated here is one typical problem that can occur during multiple fault injections: variables v_1 and v_2 are targeted at locations l_1 and l_2 respectively, and v_2 depends on v_1. Now, a fault injection in v_2 may override the effect of the error propagating from v_1 to v_2. There are extreme solutions: (i) to inject in one location and allow for error propagation (this is the traditional approach with single bit-flip model) or (ii) to inject in every single location, potentially overriding the effect of previous fault injections in that run, which is also computationally expensive.

Thus, as fault injection at every single location is not viable and fault injection at a single location may not yield accurate results, a, possibly optimal, set of possible locations should be chosen. There is a trade-off to be made then: to inject early but potentially overriding the propagating error later during the

execution or to inject late but with reduced error propagation (this is taken to mean a lesser number of variables with corrupted values). To guide the selection of variables, two pieces of information are leveraged:

1. Locations based upon their perceived impact on the output are considered
2. Variables based upon their potential combined effect when corrupted are chosen

For the first problem, the perceived impact of a location on the output may be estimated by detecting the invariants at various locations in the software. This can be possibly achieved by approximating the predicate at various locations through the use of abstract interpretation-based techniques, as proposed in [68]. Overall, it is possible then to tag or label locations with a boolean value, with a value of 1, indicating that the location can impact on the output. This does not mean that all fault injection runs will lead to a deviation, but rather that there is some fault injection run that will lead to some problem. Here, such locations are referred to as vulnerable locations.

However, not all vulnerable locations are suitable for fault injection. For example, a vulnerable location can exist within a branch and, if a variable is selected for fault injection at that location, the variable may not be corrupted at runtime as the execution may not follow the said branch. Thus, selecting vulnerable locations that will always be reached and that are reached earlier is better since these locations will provide a higher probability of error propagation and hence of causing a system failure. Henceforth, these locations are called potential locations. These locations are referred to as potential locations since, when target variables are selected, no variable may be a chosen at the given location. They merely indicate that such locations are good candidates.

For the second problem, given a set of potential locations, variables are chosen according to the potential combined effect when corrupted.

If a variable v is not targeted, then it means that either it is dependent on a variable that has already been targeted (injecting a fault in v would overwrite the propagating error) or a variable that depends on it has been targeted. Thus, the objective is to target a small set of variables while maximising the number of possible deviations (failures).

Earlier in Chapter 5 each problem is formalised as an optimisation problem, and is shown to be NP-complete. Later in this chapter, a methodology to solve each problem is developed. Specifically, the heuristics for injection locations selection and target variables selection are developed and their usability is demonstrated.

7.1 Selecting Locations for Mulitple Fault Injections

As previously mentioned, the single fault that is traditionally assumed by SWIFI techniques has become limited as it does not take into account multiple faults and their possible interactions during executions. Few works have addressed multiple fault injections [43, 79] and recently [103, 163], which addressed the problem in a more systematic way. Aspects of mutation testing [70], through the use of higher order mutants, can be considered as instances of multiple fault injections.

Although there are works that studied the impact of multiple faults [1, 9, 103] in the form of double faults, there is a dearth of work that addresses the problem selecting the injection locations for multiple fault injections in a systematic way. Lu et al. [103], to test the applicability of their fault injection tool, considered double faults models. Similar to other work that adopted the L_1C_2 models, they random flip two bits in a random target location. However, for the L_2C_1 fault, they selected the first target location randomly and use a time window, in terms of execution cycle, to select the second injection location. They used time

windows between 1 to 4. Similar to Lu et al. [103]'s work, the work here injects into register, and considered the L_1C_2 and L_2C_1 as two of the specific cases of the L_nC_m fault model evaluated in this chapter. However, the work here differs from the work in Lu et al. [103], not only in objective but in the approach selecting the fault injection points. The work, uses the proposed framework to systematically select candidate variables to target.

Perhaps the work closest in spirit to that proposed here is that in [72], where the authors develop a fault injection tool for cloud environment where multiple failures are assumed to occur. To handle the exponential fault space, the authors proposed pruning strategies that are based on static properties of specific systems such as HDFS. Also, the fault type assumed in [72] is different to what is assumed in this work, i.e., this work assumes bit-flips whereas Joshi et al. [72] assume component failures. Further, the pruning strategies are developed at an early stage whereas in this chapter, pruning strategies are developed at "runtime".

7.2 Injection Location Selection (ILS)

One disadvantage of SWIFI is that of *intrusiveness.* The impact of the use of multiple fault injections on software systems, such as control software for real-time embedded systems, will cause a high temporal overhead, causing deviation in the program's behaviour. Thus, the trade-off is that, on one hand, choosing several potential fault injection locations may help to uncover vulnerabilities but, on the other hand, the intrusiveness may cause the program to significantly deviate from its original behaviour.

Thus, for multiple fault injections to work, this thesis seek to reduce the number of fault injections to be performed while increasing the effect of each one, i.e., it is counter-productive to inject into two variables where the respective effects are

cancelled. To bound the number of possible fault injections, the number of potential locations selected for injection are minimised with the aim of maximising their effect. This is done by requiring the selected locations to be close to each other, resulting in increased error propagation (i.e., a potentially greater number of variables with corrupted values). This thesis capture this notion of location closeness by a parameter, which is called *amplification*. However, selecting such a set of potential locations at which to inject faults is very difficult.

7.2.1 Heuristic for ILS

To develop a heuristic for injection location selection, the concept of *dominators and dominance* is adopted [92, 126]. First, the dominators concept will be briefly outlined before explaining the heuristic. In a CFG $G = \langle V, v_0, A \rangle$, where v_0 is the entry node, a node M is said to *dominate* a node N, (M *dom* N), if and only if every path from v_0 to N goes through M. Based on this basic dominance concept, several other concepts can be developed, namely:

- By definition the dominance relationship is:
  - *Reflexive*, i.e., every node dominates itself. Thus (N *dom* N) is always true. Note a node is said to *trivially dominates* itself.
  - *Transitive*, i.e, if a node M dominates a node N and N dominates a node R, then M dominates R. Thus if (M *dom* N) and (N *dom* R) then (N *dom* R) is always true.
  - *Anti-symmetric*, i.e., if a node M dominates a node N and N dominates node M, this implies that $M = N$. Thus if (M *dom* N) and (N *dom* M) then $M = N$ is true.
- M *strictly dominates* N, (M *sdom* N), if and only if (M *dom* N) and $M \neq N$. Thus, a node cannot strictly dominates itself, i.e., (N *sdom* N) is never true.

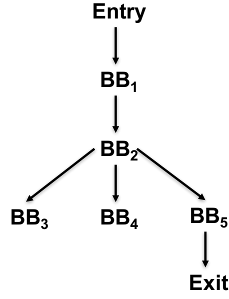
- A node M *immediately dominates* a node N, (M *idom* N), if and only if (M *sdom* N) and there does not exist a node D such that (D *sdom* N) and (M *sdom* D). Thus if a node has more than one dominator, there is always a unique *"nearest"* strict dominator called its *immediate dominator*. Note all nodes except the entry node have immediate dominators.

- The *dominance frontier* of a node M, DF_M, is the set of all nodes that are immediate successors to nodes dominated by M, but which are not themselves strictly dominated by M. This means the dominance frontier of M is the set of nodes where M's dominance stops, i.e., the set of nodes where M lies only in some of the paths in G from v_0 to the these nodes. Thus $DF_M = \{Z \mid (N\ idom\ Z) \wedge (M\ dom\ N) \wedge \neg(M\ sdom\ Z)\}$. Note a node can be in its own dominance frontier.

- The *dominator tree* of G $DomTree_{(G)}$ is a tree created using immediate dominators, where a parent node has as its children the nodes it immediately dominates and the entry node v_0 is the root of the tree. Thus $M \rightarrow N$ exists in the $DomTree_{(G)}$ if and only if (M *idom* M). Note: A node in a dominator tree dominates all its descendants in the tree, and immediately dominates all its children.

It should be mentioned that dominators reveals which basic block in a CFG must be executed prior to a block N, they also reveal blocks that are not always executed. The complexity of generating the dominator tree is $O(|U|^2)$. The dominator tree for the example CFG shown in Figure 3.1c is illustrated in Figure 7.1a; Figure 7.1b depicts the dominance relationship for the dominator tree. The coloured nodes depicts dominator nodes, i.e., nodes that strictly dominate one or more nodes.

Given a CFG $G_P = \langle V, v_0, A, W, \Phi \rangle$ of program P with amplification factor $\mathbb{A}$, the heuristic works as in Heuristic 7.1: first, the dominator tree of G_P, $DomTree_{G(P)}$, is generated, and the edges are labelled with the corresponding



(a) Dominator Tree

Basic Block	Immediate Dominator	Dominates	Dominance Frontier
Entry	None	{Entry, BB_1, BB_2, BB_3, BB_4, BB_5, Exit}	{Ø}
BB_1	Entry	{BB_1, BB_2, BB_3, BB_4, BB_5, Exit}	{Ø}
BB_2	BB_1	{BB_2, BB_3, BB_4, BB_5, Exit}	{Ø}
BB_3	BB_2	{BB_3, Exit}	{BB_2, BB_4, BB_5}
BB_4	BB_2	{BB_4}	{BB_2, , BB_5}
BB_5	BB_2	{BB_5, Exit}	{Ø}
Exit	BB_5	{Exit}	{Ø}

(b) Dominance Relationships

Figure 7.1: Example of a dominator tree for a CFG and its corresponding dominance relationships.

weights from G_P. Each node are tagged with their dominance frontier as dictated by G_P. The set of possible locations U^l is initially set to V. Then, all leaf nodes of $DomTree_P$ are removed from U^l, and any node n that is not deemed vulnerable, i.e., any node n of $DomTree_P$ with a non-empty dominance frontier ($DF_n \neq \{\emptyset\}$), were also removed from U^l.

The set U^l represents the set of potential fault injection locations. It does not represent the actual locations where faults will be injected, but rather where faults could potentially be injected. Given that $\mathbb{A}$ is set to twice the longest distance between two successive potential locations - which can be obtained

input : $G_P = (U_P, u_0, A, W, \Phi)$
output: Set of locations/blocks
begin
1. Generate the $DomTree_{G(P)}$ of G_P, label edges in $DomTree_{G(P)}$ from G_P, tag nodes in $DomTree_{G(P)}$ with dominance frontier from G_P;
2. Initially, set U^l to be U_P;
3. Remove all leaf nodes of $DomTree_{G(P)}$ from U^l and all nodes that immediately dominate the exit node.;
4. Remove any node n in $DomTree_{G(P)}$ where $DF_n \neq \{\emptyset\}$ from U^l;
5. Remove every node u such that $\Phi(u) = 0$ from U^l;

end

Heuristic 7.1: Heuristic for Injection Location Selection (ILS)

from a software engineer, this distance is equal to twice the longest distance between two dominator nodes. Thus, the set of potential locations obtained from Heuristic 7.1 satisfies the amplification factor $\mathbb{A}$. Also, the reason the leaves of the dominator tree are removed is that injection of faults at these locations do not guarantee error propagation. The complexity of the ILS heuristic is $O(|U|^2)$.

7.3 Target Variable Selection (TVS)

Following the identification of the set of potential locations, it is necessary to identify the set of variables to target for fault injection. This means, there may be potential locations at which no fault will be injected and others where more than one fault may be injected.

To determine this target variable set, the dominator tree generated by the heuristic presented in Chapter 7.2.1 (see Heuristic 7.1) is used and transformed into a weighted graph, where a dependency graph is superimposed upon the dominator tree. In this thesis, such a dependency graph for a program P upon its dominator tree is modelled as follows: the dependency graph for P is denoted as $G_P^D = \langle U, A, U^0, W, L\rangle$, where U is the set of nodes (representing variables

of P), A representing the set of arcs, where $(u, v) \in A$ means that variable u depends on variable v, U^0 is the set of nodes with no outgoing arcs (variables which do not depend on any other variables), W is the function that returns the weight on the arcs, and L is a function (called *level*) that maps a variable to a given block in the dominator tree D_P. The dependency graph of P is extended in such a way that A is augmented to include arcs between nodes at the same level with a weight 0. The significance of this is that if there are two variables v_1, v_2 in the same block in the dominator tree, then it means that it is irrelevant if a fault is injected in v_1 first and v_2 second or vice-versa.

The challenge in selecting target variables stems from the fact that when a variable u is overlooked, then it means either that a variable v on which it depends has been selected (and selecting u will override the effect of propagating error from v to u) or a variable w that depends on v has been selected. Thus, the decision of selecting a variable is not a local one. In Chapter 6, it has been demonstrated that early injections have a potential of uncovering more vulnerabilities. As such, the TVS heuristic is made biased towards selecting variables in earlier blocks.

7.3.1 Heuristic for TVS

This Chapter presents a heuristic (See Heuristic 7.2) that returns a list of variables, together with their corresponding locations, that solves the TVS problem.

A function called *level* that maps a given variable onto the block it appears in the dominator tree is assumed. The heuristic then works as follows: A variable *block* is defined, which is set to 1 initially, to capture the block being visited in the dominator tree. The heuristic takes as input the dependency graph of the dominator tree and outputs a set of variables at given locations. It starts at level 1 of the dominator tree and considers all the nodes at that level.

```
    Input: Input weighted dependency graph G = (U, A, U^0, W, L)
    Output: Set of variables for fault injection with their location
    begin
1     block : = 1;          // keeps track of the blocks from the
      dominator tree
2     NotAllowed: set of (variable, block) init ∅;
        // keeps track of the location of the variable in the
      dominator tree
3     SelectedVars: set of (variable, block) init ∅;
       // selects the variable and its location in the program
4     while( block ≤ N )
       do {
5         forall n ∈ {m | level(m) = block}
           do {
6             if( ∃l : ((n, l) ∈ A ∧ [(l, level(l)) ∈ SelectedVars ∨
              ((l, level(l)) ∉ NotAllowed)]) )
               then:
7                 NotAllowed := NotAllowed ∪ {(n, level(n))};
8             elseif( ∃k : ((k, n) ∈ A ∧ (level(k) > level(n))) )
               then:
9                 SelectedVars := SelectedVars ∪ {(n, level(n))};
          } od
10        block + +;
      } od
    end
```

Heuristic 7.2: Heuristic for Target Variables Selection

A variable n at a given location x is not selected (i.e., $(n, x) \in NotAllowed$) if there is a variable l on which it depends has already been selected as a target variable for fault injection or l is not allowed (i.e., injecting in v can override the impact of the injection in l or error propagating from l) . Variable n is otherwise selected (i.e., $(n, x) \in SelectedVars$) if there is at least one variable k that depends on it. The complexity of the TVS heuristic is $O(|U|^2)$.

7.4 Case Studies

Ten programs are used as case studies: Cubic, Isqrt, CRC, FFT, Dijkstra, Insert, Remove, Search, Encfile, Decfile. Description of these program and their

input data is presented in Chapter 3.3. Fault injection experiments are performed to evaluate the viability of the proposed injection locations framework. LLFI (See Chapter 3.4.2) is used for all experiments. The proposed approach focuses on efficient selection of injection location. However, even with the heuristic proposed, there is still an exponential number of bits to flip. To keep the experiments tractable the work presented here only consider flipping maximum of four bits. Thus only single, double, triple and quadruple fault models as described in Chapter 3.2 are adopted for the evaluation of the case studies. Henceforth in this chapter, Multiple Fault Injections (MFI) is assumed to mean double bits or triple bits or quadruple bits fault injection, unless specified otherwise.

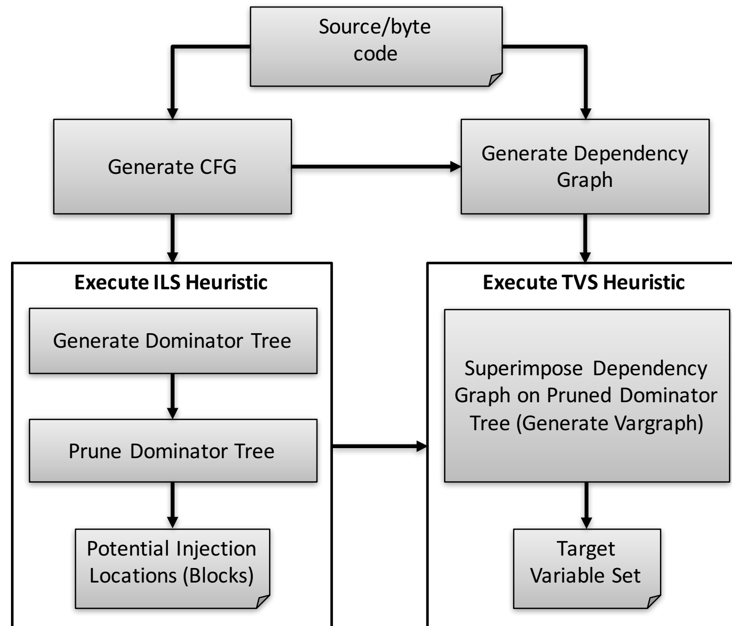


Figure 7.2: An overview for the execution of the proposed framework to select efficient target variables.

Figure 7.2 summarises the workflow of the ILS and TVS heuristics (for identifying injection locations) for the L_nC_m fault model .

7.5 Experiment Setup

The experimental procedure adopted was to address the following research questions:

- Does MFI in variables identified by the TVS heuristics uncover more vulnerabilities than MFI in interface variables?
- Does MFI in variables identified by the TVS heuristics uncover more vulnerabilities than MFI in randomly selected variables?
- Does MFI in variables identified by the TVS heuristics uncover more vulnerabilities than injecting single bit-flip (L_1C_1) faults in the same variables?
- How does MFI as multiple bit-flips faults in a single variable or MFI as single bit fault in multiple variables affect the the fault injection outcome?

The fault injections experiments performed focused on processor faults that impact on the program state by altering the content of ISA registers. Henceforth, variables denote states defined by the registers. Fault injection location is defined according to the assumed fault model under investigation as follows:

- $\mathbf{L_1C_1}$: Single bit-flip error in a single location is considered, i.e., one bit is flipped in a single location.
- $\mathbf{L_1C_2}$: Double bit-flips error in a single location is considered, i.e., two bits are flipped in a single location.
- $\mathbf{L_1C_3}$: Triple bit-flips error in a single location is considered, i.e., three bits are flipped in a single location.
- $\mathbf{L_1C_4}$: Quadruple bit-flips error in a single location is considered, i.e., four bits are flipped in a single location.

- $\mathbf{L_2C_1}$: Double single bit-flip error in a pair of locations is considered, i.e., two locations are chosen and a single bit is flipped in each location.
- $\mathbf{L_3C_1}$: Triple single bit-flip error in a triad of location is considered, i.e., three locations are chosen and a single bit is flipped in each location.
- $\mathbf{L_4C_1}$: Quadruple single bit-flips error in a quad of location is considered, i.e., four locations are chosen and a single bit is flipped in each location.
- $\mathbf{L_2C_2}$: Double bit-flips error in a pair of locations is considered, i.e., Two locations are chosen and two bit is flipped in each location.

Variable selection is done randomly from internal program variables or at program interface or by using TVS heuristics. It should be mentioned that variables selected at the program interface are done exclusively at the entry point, i.e., input interface.

7.5.1 Application of the Proposed Framewwork

The heuristics presented in preceeding sections are executed for the case studies as follows:

Step 1: Executing ILS

To apply the ILS heuristic (See Heuristic 7.1), first, the CFG (as described in Chapter 3.1 is generated. Since faults are injected into registers, the CFG for the IR byte-code of the program is generated rather than for its source code. Figure 7.3 shows the CFG for Isqrt. Due to space limitations, only graphs for Isqrt are presented and adopted to provide an overview of the execution flow of the heuristics.

Next, the generated CFG is converted into its dominator graph by applying the dominance concept defined earlier on in this chapter. The dominator tree for Isqrt is shown in Figure 7.4. Then, the set of potential injection locations, U_l,

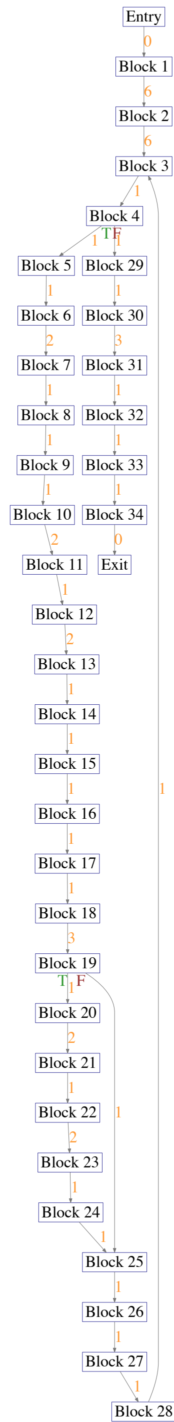


Figure 7.3: (Extended) CFG for Isqrt

is set to contain all nodes of the dominator graph. Thus:

$$U_l = \{block_1, \dots, block_{34}\}$$

Next, $block_{24}$ and $block_{28}$, being leaf nodes, are taken out from U_l, nodes dominating the exit block is also removed, in this case $block_{34}$. Finally, all nodes with non-empty dominance frontier set are taken out U_l, thus $block_5$ through $block_{23}$ and $block_{25}$ through $block_{27}$ are removed and the following set of blocks is returned.

$$U_l = \{block_1, \dots, block_4, block_{29}, \dots, block_{34}\}$$

Step 2: Execute TVS

To apply the TVS heuristic (see Heuristic 7.2), the dependency graph for the program is superimposed on its dominator tree generated in Step 1. The superimposed graph is condensed to only include nodes (blocks) returned by the ILS heuristics.

In Figure 7.5 the superimposed dependency graph over the dominator tree (of blocks set returned by ILS heuristics) for Isqrt is depicted.

Figure 7.6 simplifies the superimposed dependency graph as a variable graph (vargraph) and Heuristic 7.3 shows the Heuristic for generating a vargraph. The complexity of this heuristic is $O(|U|^2)$.

Then, the TVS heuristic is applied on the vargraph of Isqrt to obtain the MTVS. MTVS is initialised to be null and the process is started in $block_1$. All variables in $block_1$ are added to the MTVS, since the MTVS is empty, i.e., there are no variable in MTVS that any of $block_1$ variables are dependent on. Thus variables %1, %2, %a, %r, %e and %i are added to MTVS. Moving to $block_2$, because variables %1, %2, %a, %r, %e and %i are dependent on %1, %2, %a, %r, %e

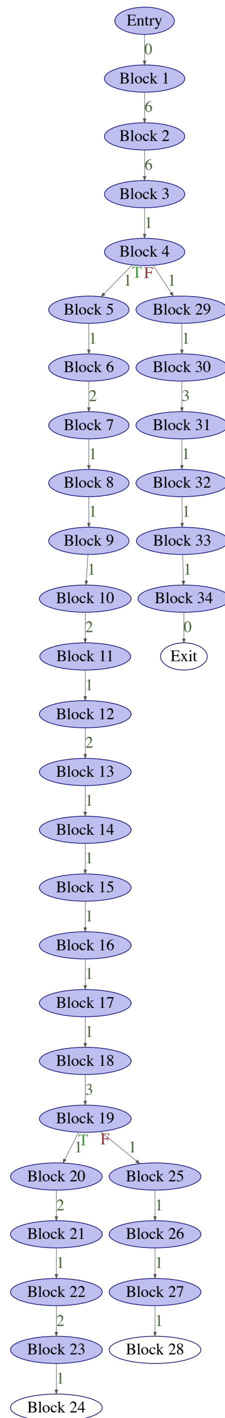


Figure 7.4: Dominator tree for Isqrt

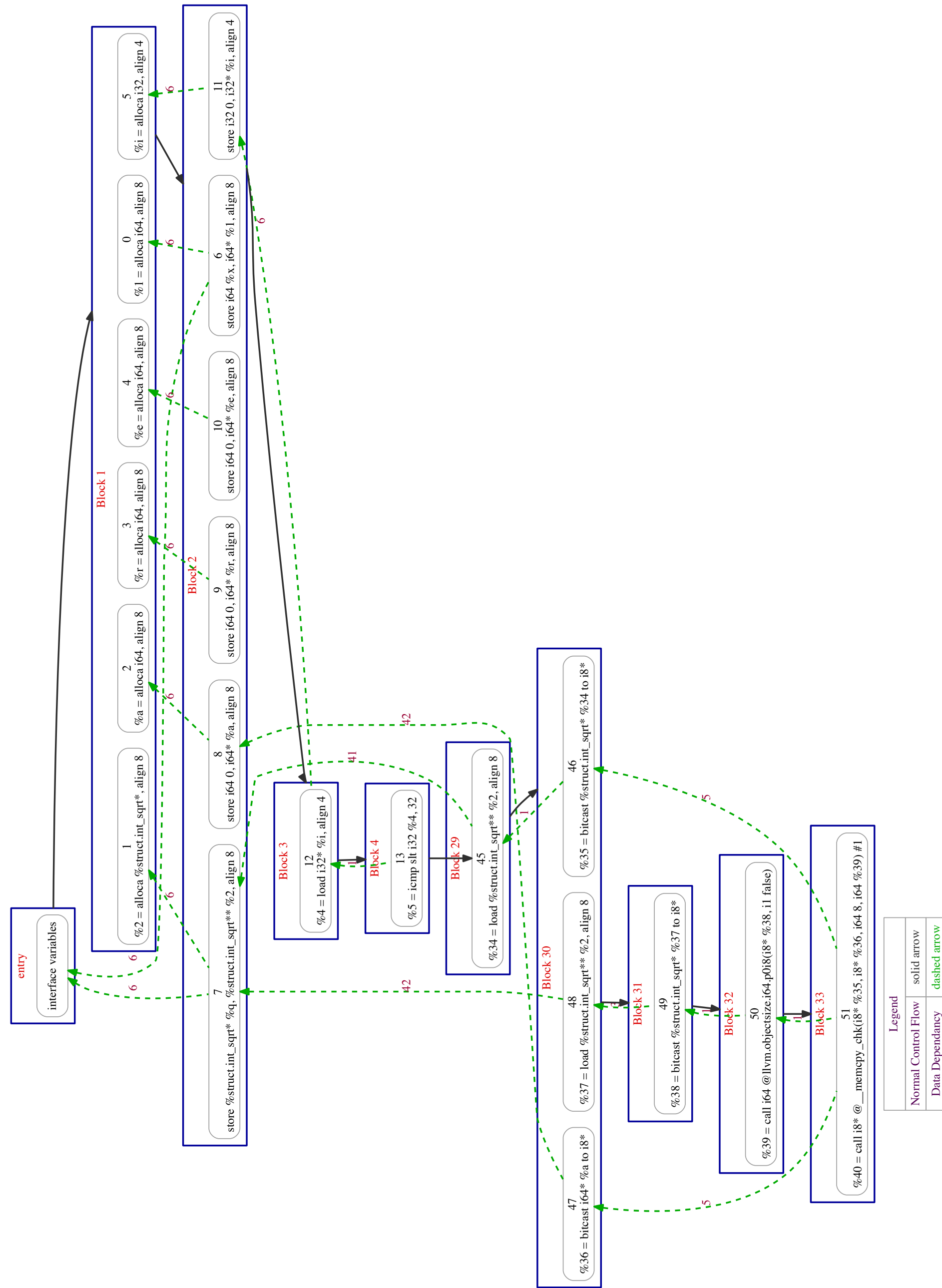


Figure 7.5: Dependency graph superimposed on dominator tree for Isqrt over its potential injection location set.

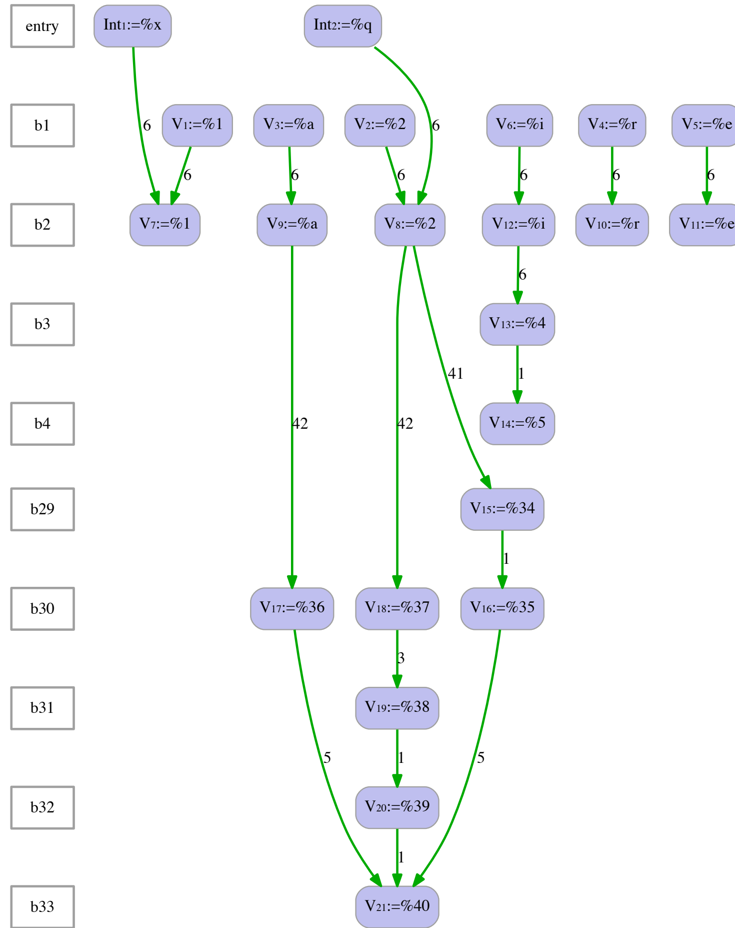


Figure 7.6: Variable Graph for Isqrt

and $\%i$ in $block_1$ respectively, they are not added to MTVS. However, variables $\%1$ and $\%2$ are dependent on the interface variables x and q respectively. Thus, as x and q are not dependent on any variable in MTVS they are added to the set. Moving through the remainder blocks, no eligible variable is found for MTVS as all variables encountered are dependent on at least one variable contained in MTVS.

Thus executing the TVS heuristic on Isqrt returned the following variables at the following locations:

```
    Input: Dominator dependency graph G = (U, A, U^0, W, L)
    Output: Variable graph G' = (V', A', U_0, W')
    begin
1       block, A', V' := 1, ∅, ∅ ;   // keep track of the blocks from
        the dominator tree
2       while( block ≤ N )
         do {
3           forall n ∈ {m | level(m) = block}
             do {
4               if( ∃l : ((l, n) ∈ A) )
                 then:
5                   V' := V' ∪ {n};
                fi
            } od
6           if( block = 1 )
             then:
7               U_0 := V';
            fi
8           block ++;
        } od
9       forall m ∈ V'
         do {
10          if( ∃n ∈ V' : (m, n) ∈ A )
             then:
11              A' := A' ∪ {(n, m)};
            fi
        } od
12      W' := W;
    end
```

Heuristic 7.3: Algorithm to obtain a *variable* graph

$$\{(\%1, block_1), (\%2, block_1), (\%a, block_1), (\%r, block_1),$$
$$(\%e, block_1), (\%i, block_1), (\%x, block_2), (\%q, block_2), \}$$

It should be mentioned that in all the case studies, all interface variables are returned as subsets of the variables set returned by the TVS heuristic.

7.5.2 Experiment Procedure

It should be mentioned that the programs are instrumented using the instrumentation process explained in Chapter 6.2.1, however, the variables are chosen using the three different variable selection methods and the errors injected are based on the different fault models adopted in this chapter. It should also be observed execution flow, campaign and experiment are as defined in Chapter 6.2.1.

To answer the four research questions posed earlier in this chapter, a number of fault injection experiments into a number of different variables (or combinations of variables) identified by different selection methods was conducted. Before running these experiments, internal program variables were selected randomly for the different target programs. Table 7.1 presents the total number of variables chosen for the three different variable selection methods.

Table 7.1: Number of target variables selected for the different target programs

Programs	**Variable Selection Method**			**Program Variables (Total #)**
	Interface	*Heuristic*	*Random*	
CRC	4	11	6	38
FFT	7	33	17	212
Dijkstra	4	7	4	70
Insert	2	10	5	76
Remove	2	13	7	277
Search	3	9	5	60
Encfile	5	15	8	105
Decfile	6	21	11	126
Cubic	2	8	4	171
Isqrt	6	19	10	39

For each execution flow, fault injection at locations corresponding to interface variables was executed at source registers of Store instructions. All other injections were done into destinations registers of their respective locations. Variables are grouped according to their selection method. For each group of variables, errors injections consisting of double, triple and quadruple locations, were done in every pairwise, triadwise and quadwise combination of variables, respectively. For each campaign, 1000, 2000, 3000 and 4000 random experiments were con-

ducted for each single errors, each double errors, each triple errors and each quadruple errors, respectively. A total of 53, 163, 000 fault injections was conducted (See Table 7.2).

Table 7.2: Total number of fault injection experiments conducted over all target programs

Per Fault Model	
L_1C_1	2340000
L_1C_2	4023000
L_1C_3	6363000
L_1C_4	8703000
L_2C_1	5994000
L_3C_1	7020000
L_4C_1	9360000
L_2C_2	9360000
Per Variable Selection Method	
Interface	8487000
Heuristic	30222000
Random	14454000

To better understand the profile of the software the failure scheme defined in Chapter 3.4.3 is adopted to categorise the outcomes of the fault injection experiments.

7.6 Evaluation of the Case Studies

This section sought to answer the research question introduced earlier in this chapter (see Chapter 7.5). First, in the following section, the impact of variable selection on experiments outcome is discussed and the various selection methods used in the experiments are compared, as reported in Chapter 7.5. The experiments outcome shows:

1. Interface variables are subset of TVS of variables, and injections in interface variables cause more SDCs.

2. Injections in the full set of TVS cause more severe failure,

3. Injections in random internal variables show no consistent failure pattern,

4. Injecting L_nC_m faults causes more failures than injecting single faults,

5. Injecting L_1C_m faults causes more SDCs and the higher the number of corruptions the higher the proportion of SDCs,

6. injecting L_nC_1 faults causes more severe failures and the higher the number of corruptions the higher the proportion of SDCs.

This and following sections further analyse and discuss the impact of fault model on experiments outcome. Analyses will be based on the data shown in Tables 7.3–7.10 and in Figures 7.7a–7.7c.

Tables 7.3–7.10 depicts the percentage of error sensitivity for the different programs for each fault model under consideration. Each row shows the error sensitivity to a particular failure class for a particular variable selection method.

Figures 7.7a–7.7c shows the average error sensitivity for the three variable selection methods across the different fault models. In each figure, the vertical axis denotes the average error sensitivity to the failure classes (over all the programs) and the horizontal axis depicts the different fault models.

Table 7.3, Table 7.4, Table 7.5, Table 7.6, Table 7.7, Table 7.8, Table 7.9 and Table 7.10 shows the results under L_1C_1, L_1C_2, L_1C_3, L_1C_4, L_2C_1, L_3C_1, L_4C_1, and L_2C_2, respectively. Figure 7.7a, Figure 7.7b, and Figure 7.7c shows the results for variables selected randomly, variables selected at the interface and variables selected by the TVS heuristic. In this analysis the variables selected by the three methods will be referred to as *random variables*, *interface variables* and *TVS variables*.

Table 7.3: Average error sensitivity distributions for different programs for L_1C_1

Program	**No Impact (%)**			**Exception (%)**		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	8.95	7.17	3.99	55.64	30.22	26.17
dijkstra	33.85	20.66	10.94	0.00	0.00	0.00
fft	10.74	9.02	4.50	50.94	31.43	22.23
search	64.48	36.50	10.97	0.00	0.00	0.00
insert	40.78	26.94	6.89	0.00	0.00	0.00
remove	9.98	7.99	2.76	0.00	0.00	0.00
encfile	14.09	9.49	3.42	43.80	26.69	22.46
decfile	14.13	9.57	3.38	43.92	26.90	22.20
isqrt	12.50	2.70	6.71	23.90	31.63	22.22
cubic	17.50	15.62	15.32	24.90	31.53	23.82

Program	**SDC (%)**			**Time Out (%)**		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	12.06	9.66	5.37	4.28	3.43	1.91
dijkstra	6.77	42.15	30.21	0.00	0.00	0.00
fft	13.59	11.62	5.67	3.92	3.56	1.62
search	12.90	37.23	18.18	0.00	0.00	0.00
insert	16.31	43.97	19.02	0.00	0.00	0.00
remove	19.97	48.88	22.89	0.00	0.00	0.00
encfile	23.74	17.06	8.07	3.37	3.03	1.64
decfile	23.80	17.19	7.97	3.38	3.05	1.62
isqrt	19.10	27.33	15.22	0.20	0.00	0.00
cubic	15.60	15.52	8.51	0.00	0.10	0.20

Program	**Crash (%)**		
	Random	*Interface*	*Heuristic*
crc	19.07	49.53	62.56
dijkstra	59.38	37.19	58.85
fft	20.81	44.36	65.98
search	22.62	26.28	70.85
insert	42.91	29.09	74.10
remove	70.05	43.13	74.34
encfile	15.01	43.74	64.42
decfile	14.78	43.29	64.83
isqrt	44.20	38.34	55.96
cubic	41.90	37.24	52.25

7.6.1 Variable Selection Method Effects

Figure 7.6 shows the distribution of the fault injection outcomes over all programs, grouped by fault model. These results were obtained through injections into interface variables, random variables or TVS variables. The failure modes

Table 7.4: Average error sensitivity distributions for different programs for L_1C_2

Program	**No Impact (%)**			**Exception (%)**		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	4.55	3.54	1.68	57.95	25.00	15.23
dijkstra	24.40	1.32	2.60	0.00	0.00	0.00
fft	6.67	4.56	2.37	51.93	23.81	14.78
search	56.22	2.03	2.60	0.00	0.00	0.00
insert	31.87	0.94	1.55	0.00	0.00	0.00
remove	6.86	0.21	0.48	0.00	0.00	0.00
encfile	7.75	4.90	1.64	49.39	23.06	14.88
decfile	7.75	4.90	1.64	49.39	23.06	14.88
isqrt	5.89	0.40	2.10	17.88	27.20	15.72
cubic	7.89	18.10	15.32	22.68	16.70	9.01

Program	**SDC (%)**			**Time Out (%)**		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	8.52	6.64	3.15	1.70	1.33	0.63
dijkstra	4.85	39.80	24.28	0.00	0.00	0.00
fft	10.23	7.51	3.80	1.53	1.26	0.61
search	11.18	61.55	24.28	0.00	0.00	0.00
insert	12.67	56.94	28.88	0.00	0.00	0.00
remove	13.63	62.69	45.14	0.00	0.00	0.00
encfile	18.16	12.24	5.39	1.45	1.22	0.62
decfile	18.16	12.24	5.39	1.45	1.22	0.62
isqrt	20.68	26.70	13.41	0.00	0.00	0.00
cubic	23.28	19.50	9.81	0.00	0.00	0.80

Program	**Crash (%)**		
	Random	*Interface*	*Heuristic*
crc	27.27	63.50	79.31
dijkstra	70.75	58.88	73.12
fft	29.63	62.85	78.44
search	32.60	36.42	73.12
insert	55.45	42.12	69.58
remove	79.52	37.10	54.38
encfile	23.24	58.57	77.48
decfile	23.24	58.57	77.48
isqrt	55.54	45.70	68.77
cubic	46.15	45.70	65.07

of each fault model are calculated as the mean across all programs.

Injection in TVS Variables vs. Injection in Interface Variables

The observations from Figure 7.7b, and Figure 7.7c are as follows:

Table 7.5: Average error sensitivity distributions for different programs for L_1C_3

Program	**No Impact (%)**			**Exception (%)**		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	1.94	2.41	0.99	38.83	24.82	11.13
dijkstra	19.60	0.00	0.96	0.00	0.00	0.00
fft	3.60	3.48	1.51	36.04	23.65	10.89
search	51.04	0.00	0.96	0.00	0.00	0.00
insert	27.00	0.00	0.57	0.00	0.00	0.00
remove	5.42	0.00	0.18	0.00	0.00	0.00
encfile	3.76	3.53	0.98	37.56	24.24	11.09
decfile	3.76	3.53	0.98	37.56	24.24	11.09
isqrt	2.57	0.06	0.61	12.35	22.49	10.28
cubic	3.24	19.00	13.40	18.81	8.01	2.98

Program	**SDC (%)**			**Time Out (%)**		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	0.97	1.20	0.49	0.00	0.00	0.00
dijkstra	3.40	35.67	21.48	0.00	0.00	0.00
fft	2.70	2.33	1.03	0.00	0.00	0.00
search	8.85	58.09	21.48	0.00	0.00	0.00
insert	9.37	52.58	25.57	0.00	0.00	0.00
remove	9.40	58.09	40.84	0.00	0.00	0.00
encfile	2.35	2.35	0.86	0.00	0.00	0.00
decfile	2.35	2.35	0.86	0.00	0.00	0.00
isqrt	20.66	25.08	10.94	0.00	0.00	0.00
cubic	31.64	22.20	9.90	0.00	0.00	2.80

Program	**Crash (%)**		
	Random	*Interface*	*Heuristic*
crc	58.25	71.57	87.39
dijkstra	77.00	64.33	77.56
fft	57.66	70.55	86.57
search	40.10	41.91	77.56
insert	63.64	47.42	73.86
remove	85.18	41.91	58.98
encfile	56.34	69.88	87.07
decfile	56.34	69.88	87.07
isqrt	64.42	52.37	78.17
cubic	46.31	50.79	70.91

Overall, injection in TVS variables tend to induce more severe failure than injection at only interface variables. On the other hand, injection at interface variables are more susceptible to SDCs and Exception failures than injection in TVS variables. With the exception of L_1C_4 and L_2C_1, injections into interface variables are more probable to result in No impact than injections into the TVS

Table 7.6: Average error sensitivity distributions for different programs for L_1C_4

Program	No Impact (%)			Exception (%)		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	3.17	2.41	1.63	63.49	24.82	0.49
dijkstra	24.53	0.00	0.16	0.00	0.07	0.32
fft	5.63	3.48	1.65	56.34	23.65	0.49
search	57.21	0.01	0.18	0.00	0.11	0.32
insert	32.49	0.01	0.11	0.00	0.10	0.38
remove	6.99	0.06	0.06	0.00	0.11	0.62
encfile	6.02	3.53	1.62	60.15	24.24	0.48
decfile	6.02	3.53	1.62	60.15	24.24	0.48
isqrt	1.06	0.01	0.17	8.12	18.19	6.42
cubic	1.25	18.90	10.68	14.67	3.64	0.90

Program	SDC (%)			Time Out (%)		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	1.59	1.20	0.81	0.00	0.00	0.00
dijkstra	4.07	32.20	19.32	0.00	0.00	0.00
fft	4.23	2.33	0.84	0.00	0.00	0.00
search	9.48	54.24	19.32	0.00	0.00	0.00
insert	10.77	48.68	23.03	0.00	0.00	0.00
remove	11.59	54.21	37.33	0.00	0.00	0.00
encfile	3.76	2.35	1.41	0.00	0.00	0.00
decfile	3.76	2.35	1.41	0.00	0.00	0.00
isqrt	19.66	23.06	8.52	0.00	0.00	0.00
cubic	40.41	23.94	9.10	0.00	0.00	8.94

Program	Crash (%)		
	Random	*Interface*	*Heuristic*
crc	31.75	71.57	97.07
dijkstra	71.40	67.73	80.20
fft	33.80	70.55	97.03
search	33.30	45.64	80.18
insert	56.74	51.21	76.47
remove	81.41	45.62	61.99
encfile	30.08	69.88	96.48
decfile	30.08	69.88	96.48
isqrt	71.15	58.74	84.89
cubic	43.67	53.51	70.38

heuristic variables. There is negligible difference in time out failures induced.

Injecting L_1C_1, L_1C_2, L_1C_3, L_1C_4, L_2C_1, L_3C_1, L_4C_1 and L_2C_2 errors in TVS variables and interface variables mirrors the overall trend observed for each fault outcome.

Table 7.7: Average error sensitivity distributions for different programs for L_2C_1

Program	No Impact (%)			Exception (%)		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	9.97	14.02	3.93	59.12	8.03	4.89
dijkstra	10.76	0.00	2.18	0.00	0.00	0.00
fft	11.56	14.19	4.14	52.87	7.91	4.84
search	31.03	0.00	2.18	0.00	0.00	0.00
insert	14.36	0.00	1.33	0.00	0.00	0.00
remove	2.60	0.00	0.46	0.00	0.00	0.00
encfile	18.06	19.55	3.93	53.54	7.47	4.89
decfile	18.06	19.55	3.93	53.54	7.47	4.89
isqrt	8.10	0.00	1.50	3.30	12.60	8.19
cubic	4.50	20.68	13.30	15.60	0.00	0.00

Program	SDC (%)			Time Out (%)		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	0.30	0.51	0.14	0.00	0.00	0.00
dijkstra	7.59	36.35	13.03	0.00	0.00	0.00
fft	2.91	0.90	0.38	0.00	0.00	0.00
search	21.88	58.81	13.03	0.00	0.00	0.00
insert	20.26	53.32	15.90	0.00	0.00	0.00
remove	18.36	58.81	27.63	0.00	0.00	0.00
encfile	0.68	0.95	0.25	0.00	0.00	0.00
decfile	0.68	0.95	0.25	0.00	0.00	0.00
isqrt	56.40	13.20	6.79	0.00	0.00	0.00
cubic	19.30	17.08	9.20	0.00	0.10	0.30

Program	Crash (%)		
	Random	*Interface*	*Heuristic*
crc	30.61	77.44	91.03
dijkstra	81.65	63.65	84.78
fft	32.66	77.00	90.63
search	47.09	41.19	84.78
insert	65.38	46.68	82.76
remove	79.03	41.19	71.90
encfile	27.72	72.02	90.93
decfile	27.72	72.02	90.93
isqrt	32.20	74.20	83.52
cubic	60.60	62.14	77.20

From the results in Tables 7.3–7.10, it is deduced that: on program level, similar to the already observed failure trend, injecting faults into TVS variables causes higher crash rate than injection in interface variables, with observable percentage difference ranging from 13.03% to 45.00%, 14.24% to 36.70%, 13.23% to 35.65%, 12.47% to 34.54%, 9.32% to 43.59%, 5.73% to 36.02%, 0.90% to

Table 7.8: Average error sensitivity distributions for different programs for L_3C_1

Program	**No Impact (%)**			**Exception (%)**		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	0.10	1.05	0.05	52.99	4.22	0.50
dijkstra	0.68	0.00	0.12	8.49	7.55	0.99
fft	2.60	2.62	0.07	50.03	8.72	0.50
search	3.10	1.30	0.17	23.88	25.98	0.99
insert	2.25	1.22	0.13	24.99	24.47	1.23
remove	1.04	4.38	0.15	17.93	8.77	2.37
encfile	0.20	2.89	0.05	52.86	7.71	0.50
decfile	0.20	3.11	0.05	53.09	8.29	0.50
isqrt	5.53	0.00	0.26	2.25	5.25	1.43
cubic	2.40	9.51	2.47	8.32	0.00	0.00

Program	**SDC (%)**			**Time Out (%)**		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	0.10	1.05	0.05	0.00	0.00	0.00
dijkstra	8.15	3.02	1.74	0.00	0.00	0.00
fft	2.60	2.62	0.07	0.00	0.00	0.00
search	24.12	11.69	1.79	0.00	0.00	0.00
insert	25.24	11.01	2.22	0.00	0.00	0.00
remove	18.11	3.95	4.29	0.00	0.00	0.00
encfile	0.25	3.85	0.09	0.00	0.00	0.00
decfile	0.25	4.15	0.09	0.00	0.00	0.00
isqrt	38.50	5.50	1.19	0.00	0.00	0.00
cubic	10.29	7.86	1.71	0.00	0.05	0.06

Program	**Crash (%)**		
	Random	*Interface*	*Heuristic*
crc	46.81	93.67	99.40
dijkstra	82.68	89.43	97.15
fft	44.78	86.05	99.35
search	48.90	61.03	97.06
insert	47.52	63.29	96.42
remove	62.92	82.90	93.20
encfile	46.69	85.55	99.36
decfile	46.46	84.45	99.36
isqrt	53.72	89.26	97.12
cubic	78.99	82.58	95.76

19.95%, 1.49% to 36.46% for L_1C_1, L_1C_2, L_1C_3, L_1C_4, L_2C_1, L_3C_1, L_4C_1 and L_2C_2 errors, respectively .

While there is not much difference in proportion of time out rate, injection at interface variables are more prone to No impact, exception failures and SDCs than injection in TVS variables.

Table 7.9: Average error sensitivity distributions for different programs for L_4C_1

Program	**No Impact (%)**			**Exception (%)**		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	0.20	0.20	0.40	0.00	0.00	0.00
dijkstra	0.00	0.00	0.20	0.00	0.00	0.00
fft	0.63	0.80	0.40	0.00	0.00	0.00
search	0.00	0.00	0.20	0.00	0.00	0.00
insert	0.00	0.00	0.12	0.00	0.00	0.00
remove	0.00	0.00	0.05	0.00	0.00	0.00
encfile	1.64	0.37	0.40	0.00	0.00	0.00
decfile	1.74	0.38	0.40	0.00	0.00	0.00
isqrt	3.44	0.00	0.04	1.40	1.84	0.22
cubic	1.13	3.17	0.39	3.92	0.00	0.00

Program	**SDC (%)**			**Time Out (%)**		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	1.58	2.20	0.20	0.00	0.00	0.00
dijkstra	1.58	2.20	1.10	0.00	0.00	0.00
fft	5.05	8.84	0.20	0.00	0.00	0.00
search	16.72	15.79	1.10	0.00	0.00	0.00
insert	13.84	9.28	1.37	0.00	0.00	0.00
remove	11.81	15.79	2.71	0.00	0.00	0.00
encfile	16.45	8.22	0.35	0.00	0.00	0.00
decfile	17.35	8.64	0.35	0.00	0.00	0.00
isqrt	13.33	7.51	0.18	0.00	0.00	0.00
cubic	4.04	17.43	0.27	0.00	0.02	0.01

Program	**Crash (%)**		
	Random	*Interface*	*Heuristic*
crc	98.22	97.60	99.40
dijkstra	98.42	97.80	98.70
fft	94.32	90.36	99.40
search	83.28	84.21	98.70
insert	86.16	90.72	98.50
remove	88.19	84.21	97.24
encfile	81.91	91.40	99.25
decfile	80.91	90.99	99.25
isqrt	81.83	90.64	99.56
cubic	90.92	79.39	99.34

One possible reason for injections in interface variables leading to more SDCs, and leading to severe failure less frequently may likely be faults in one or more of these variables only gets activated in a looping structure. We surmise that, should this hold true, error propagation is limited to a part of the state only when faults are injected in these structures, thereby not disturbing the state

Table 7.10: Average error sensitivity distributions for different programs for L_2C_2

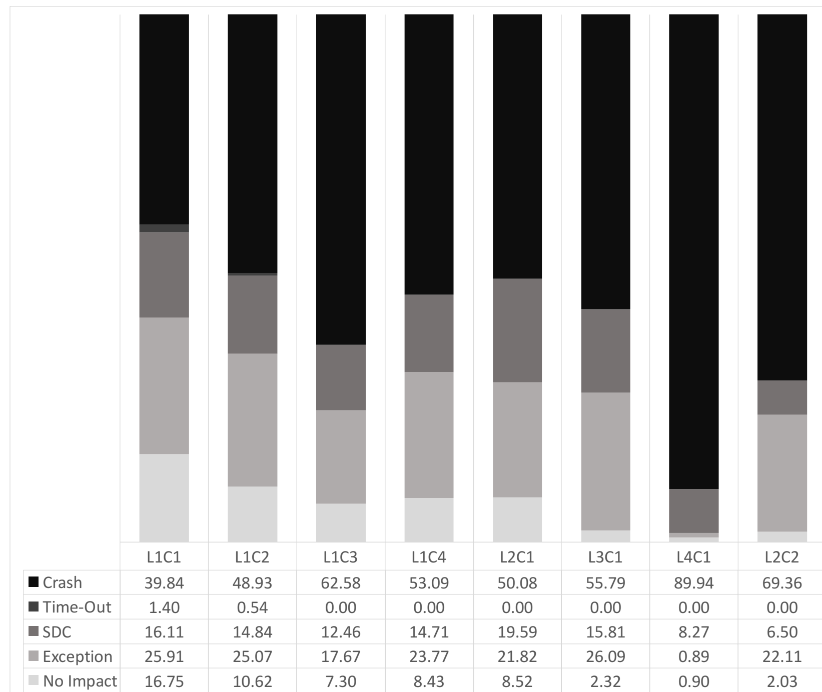
Program	No Impact (%)			Exception (%)		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	0.08	0.17	0.07	69.38	1.83	0.20
dijkstra	4.11	0.00	0.37	0.00	0.00	0.00
fft	3.12	0.26	0.08	60.93	1.82	0.20
search	14.95	0.00	0.38	0.00	0.00	0.00
insert	5.96	0.00	0.23	0.00	0.00	0.00
remove	0.98	0.00	0.09	0.00	0.00	0.00
encfile	0.16	0.25	0.07	69.24	1.81	0.20
decfile	0.16	0.25	0.07	69.24	1.81	0.20
isqrt	1.58	0.00	0.01	0.64	0.53	0.03
cubic	0.49	0.74	0.05	1.69	0.00	0.00

Program	SDC (%)			Time Out (%)		
	Random	*Interface*	*Heuristic*	*Random*	*Interface*	*Heuristic*
crc	0.08	0.17	0.07	0.00	0.00	0.00
dijkstra	5.29	28.25	9.21	0.00	0.00	0.00
fft	3.12	0.26	0.08	0.00	0.00	0.00
search	19.22	12.87	6.63	0.00	0.00	0.00
insert	15.32	10.56	8.17	0.00	0.00	0.00
remove	12.61	12.87	11.29	0.00	0.00	0.00
encfile	0.20	0.82	0.12	0.00	0.00	0.00
decfile	0.20	0.86	0.12	0.00	0.00	0.00
isqrt	6.11	24.05	6.82	0.00	0.00	0.00
cubic	1.75	45.45	9.68	0.00	0.00	0.00

Program	Crash (%)		
	Random	*Interface*	*Heuristic*
crc	30.46	97.84	99.67
dijkstra	90.60	71.75	90.42
fft	32.84	97.67	99.65
search	65.83	87.13	92.99
insert	78.72	89.44	91.60
remove	86.41	87.13	88.62
encfile	30.39	97.11	99.62
decfile	30.39	97.08	99.62
isqrt	91.67	75.42	93.14
cubic	96.07	53.81	90.27

enough to cause a crash, but only enough to cause data failures (SDCs and exception failures).

However, it should be noted that failures observed with interface variables are a subset of the failures induced by the TVS heuristic, on the account of the



	L1C1	L1C2	L1C3	L1C4	L2C1	L3C1	L4C1	L2C2
Crash	39.84	48.93	62.58	53.09	50.08	55.79	89.94	69.36
Time-Out	1.40	0.54	0.00	0.00	0.00	0.00	0.00	0.00
SDC	16.11	14.84	12.46	14.71	19.59	15.81	8.27	6.50
Exception	25.91	25.07	17.67	23.77	21.82	26.09	0.89	22.11
No Impact	16.75	10.62	7.30	8.43	8.52	2.32	0.90	2.03

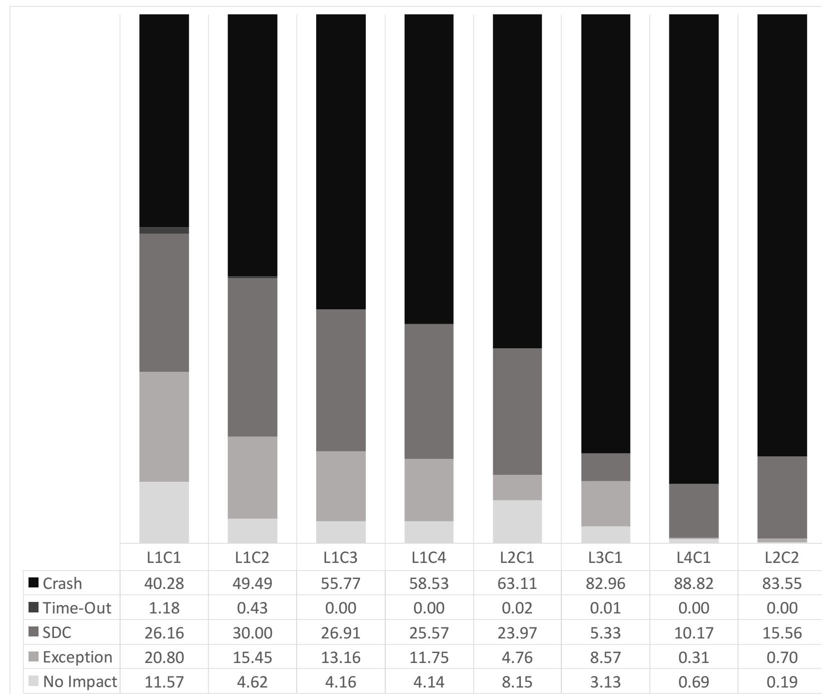
(a) Random

Figure 7.7: Average error sensitivity distribution over all target programs for different variable selection methods.

interface variables are contained in the TVS heuristic variables. This means, the failures uncovered by injections in interface variables are a subset of failures uncovered by injections in the TVS variables. The trend is noticed across the different fault models. Henceforth, interface variables and interface variables subset will be used interchangeably.

Injection in TVS Variables vs. Injection in Random Variables

First the overall observations shown in Figure 7.7a, Figure 7.7b and Figure 7.7c are presented: Very negligible difference between time out failures are observed. While proportions of Exception failures are higher when injecting in random program variables, the proportions of crash failures are $\approx 41\%$ higher when injecting in TVS variables. However, considering interface as a subset of TVS variables and comparing injections into them with injections in random vari-



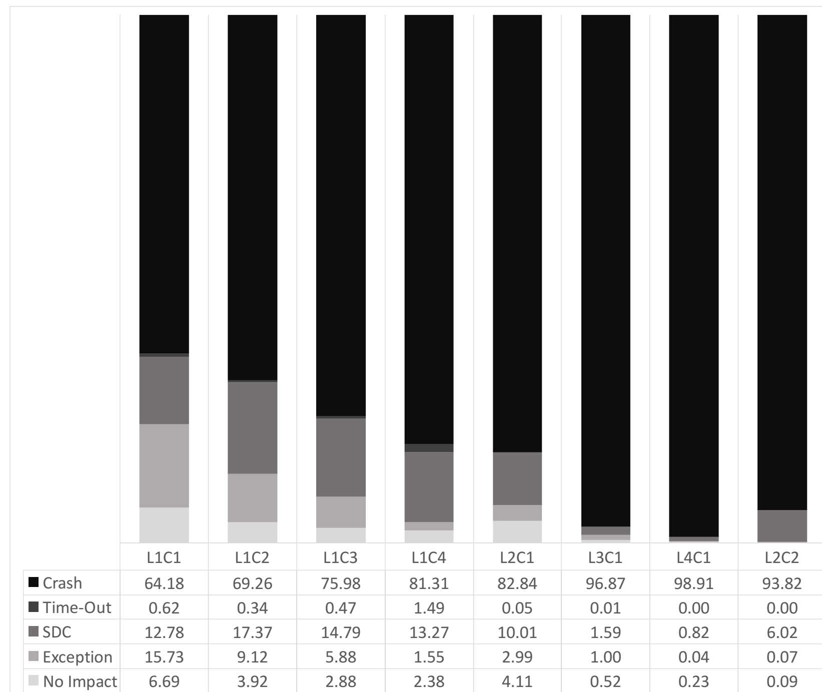
	L1C1	L1C2	L1C3	L1C4	L2C1	L3C1	L4C1	L2C2
Crash	40.28	49.49	55.77	58.53	63.11	82.96	88.82	83.55
Time-Out	1.18	0.43	0.00	0.00	0.02	0.01	0.00	0.00
SDC	26.16	30.00	26.91	25.57	23.97	5.33	10.17	15.56
Exception	20.80	15.45	13.16	11.75	4.76	8.57	0.31	0.70
No Impact	11.57	4.62	4.16	4.14	8.15	3.13	0.69	0.19

(b) Interface

Figure 7.7: Average error sensitivity distribution over all target programs for different variable selection methods.

ables, The most difference observed being ≈ 13% higher crash rate for injections in interface variables, and ≈ 7% in favour of injections in program variables, under L_3C_1 and L_1C_3 respectively. Similarly, random internal injections are mostly more prone to exception failures than interface injections.

Observations on fault model level: the most noticeable difference is in the rate of crash failures ranging from 13% to 41%, the lowest and highest difference are observed under L_1C_3 and L_3C_1 errors respectively. There is almost no difference in time out failure rate. Injections in random program variables are more prone to exception failure and more probable to result in No impact than injections in TVS variables. Similar trend is observed for injections into a interface variable compared to injections in random variables. Injections into TVS variables is often slightly more resilient to SDCs. However, injections into the interface variables subset is more sensitive to SDCs.



	L1C1	L1C2	L1C3	L1C4	L2C1	L3C1	L4C1	L2C2
Crash	64.18	69.26	75.98	81.31	82.84	96.87	98.91	93.82
Time-Out	0.62	0.34	0.47	1.49	0.05	0.01	0.00	0.00
SDC	12.78	17.37	14.79	13.27	10.01	1.59	0.82	6.02
Exception	15.73	9.12	5.88	1.55	2.99	1.00	0.04	0.07
No Impact	6.69	3.92	2.88	2.38	4.11	0.52	0.23	0.09

(c) TVS Heuristic

Figure 7.7: Average error sensitivity distribution over all target programs for different variable selection methods.

Observations on program level: the following can be inferred from Tables 7.3–7.10: while injection into TVS variable is more prone to crashes than injection into random variables (with difference ranging from $\approx 0.18\%$ to $\approx 69.22\%$), injections in random variables are often more prone to SDCs (with most observed difference of $\approx 11\%$) compared to injections in the full set of TVS variables and consistently more resilient to the same SDCs when compared to the interface variables subset. However, under certain errors for certain program injection into the full set of TVS errors is more prone to SDCs than injections into random variables. For example, under L_1C_2 for remove (patricia), injecting in full set of TVS variables induced $\approx 32\%$ higher SDCs and for Isqrt under L_2C_2 errors, injecting into random variables observed $\approx 50\%$ higher SDC rate.

We conjecture, the tendency for injections in TVS variables to induce more severe failure than injections in randomly selected program variables may be

explained by the TVS heuristic bias towards selecting target locations that will always be reached during normal program execution. This means faults inserted in TVS variables will mostly be activated, i.e will be read by another instruction in the program, except in instances where another activated fault has either change the flow of execution or cause the program to prematurely terminate.

7.6.2 Fault Model Effects

At a glance, L_2C_1, L_3C_1, L_4C_1 and L_2C_2 errors seem to consistently cause more severe failures than L_1C_4, L_1C_3, L_1C_2 and L_1C_1 errors, they also tend to be less susceptible to exception faults and less prone to SDCs. There is almost no difference between the eight models for time out failures. The multiple fault models induces less benign faults than the single fault model. The proportion of no impact faults is almost constant as corruption increases for the multiple corruptions in single location fault models (L_1C_2, L_1C_3 and L_1C_4) and decreases with corruption increase for the single corruption across multiple locations (L_2C_1, L_3C_1 and L_4C_1) and for L_2C_2.

$\mathbf{L_1C_1}$ faults vs. $\mathbf{L_nC_m}$ faults

Injecting L_nC_m faults cause higher crashes than injecting L_1C_1 faults, irrespective of variable selection method or fault type. Most crash rate difference are observed under L_4C_1 faults, $\approx 50\%$ for random variables, $\approx 49\%$ for interface variables subset and $\approx 35\%$ for TVS variables full set. Whereas the least observed rate difference are under L_1C_2 errors, $\approx 9\%$ for random variables and interface variables subset, and $\approx 5\%$ for TVS variables full set.

Proportion of exception failures is lower under L_nC_m than under L_1C_1 faults, irrespective of variable selection method. No impact rates observed under L_nC_m

are almost halved, in some cases more, as compared to L_1C_1 errors. While SDCs are less under errors injected across multiple locations (L_nC_1 and L_2C_2) than under L_1C_1 errors, SDCs are slightly more under multiple errors injected into a single variable than under L_1C_1 errors, for interface variables subset and TVS variables (full) set. Most difference observed is for L_2C_1 errors, $\approx 4\%$ and $\approx 5\%$ for interface variables subset and TVS heuristic (full) set, respectively. However, for injections in random variables there is no distinct pattern in SDC rate. And there is almost no observable difference between the L_nC_m models and the L_1C_1 model for time out failure mode.

Thus, the results show that L_nC_m faults tend to cause:

1. Higher proportion of failures,
2. Higher proportion of SDCs under L_1C_m faults,
3. A higher proportion of severe failures under the L_nC_1 faults.

We conjecture that the reason for the higher proportion of crashes is due to the fact that the impact of each individual fault is amplified, causing a big enough perturbation in system state to cause a crash.

$\mathbf{L_1C_m}$ faults vs. $\mathbf{L_nC_1}$ faults vs. $\mathbf{L_2C_2}$ faults

While injecting L_nC_1 faults cause more crashes and less no impact rates than L_1C_m faults, L_1C_m causes more SDCs and exception failure rates than L_nC_1. For injections in TVS variables, L_1C_m causes negligibly higher proportion of time out failures than L_nC_1.

Proportion of crash rate increases with number of corruptions, irrespective of selection method. SDC decreases with number of corruptions for TVS variables, whereas for interface variables subset SDC rates increase with number of corruptions under L_1C_m faults and decrease with number of corruptions under

L_nC_1 faults. However, for random variables, while SDC rate under L_1C_m faults slightly varied in no definitive way and under L_nC_1 it decreases as corruption increases.

L_1C_m injections in interface variables subset causes exception rate to slightly decrease as number of corruption increases and no definitive change rate for injecting L_nC_1. Injecting L_1C_m faults in interface variables subset causes decrease in exception failure rate as number of corruptions increases, whereas injecting L_nC_1 causes slight decrease. There are no observable pattern for injecting either L_1C_m or L_nC_1 in random variables. Proportion of no impact rate decreases with number of corruptions under either L_1C_m or L_nC_1 irrespective of variable selection method.

L_2C_2 faults cause less crashes than L_4C_1 faults but more crashes than the other L_1C_m and L_nC_1 faults, irrespective of variable selection method. Injecting L_2C_2 errors in interface variables subset or the full TVS variables set cause more SDC rate than injecting multiple faults in more than two locations (L_3C_1 and L_4C_1 errors) and less SDC rates than injecting L_1C_m and L_2C_1 faults. Whereas injecting L_2C_2 faults in random variables cause less SDCs than either L_1C_m or L_nC_1 faults. Injecting L_2C_2 errors cause lower proportion of no impact rate than injecting either L_1C_m or L_nC_1, irrespective of fault variable selection method. While injecting L_2C_2 faults cause negligibly higher proportion of exception failure rate for interface variables subset and full TVS variables set than injecting L_4C_1 faults, for random variables L_2C_2 faults cause a much higher proportion of exception failure rate than L_4C_1 faults. Whereas they cause less proportion of exception failure rate in for the other types of errors. And there is almost no observable difference between the different types of models for time out failure mode.

Thus, the results show that:

1. Crash rate increase concomitantly with corruption increase,

2. SDC rate increase with number of corruption under L_1C_m faults and decrease as number of corruption increases under L_nC_1 faults,
3. L_nC_1 faults cause more crash rate and less no impact rate than L_1C_m faults,
4. L_1C_m faults cause more SDC rate and exception failure rate than L_nC_1 faults.

We conjecture that L_nC_1 faults increase error propagation in a program, thus resulting in lower no impact rat. Secondly, they amplify the effect of errors in the program thereby causing more crashes. Also, it is postulated that because L_1C_m are constricted to a single location they do cause perturbation great enough to corrupt the data but may not be great enough to induce a crash. Secondly, the nature of the L_nC_m fault model, prohibits the selection of variables along the same path, as such variable location combinations that may potential amplify the effect of SDCs are overlooked. This is also postulated to account for the reason why the L_1C_m models are causing higher SDC rate that the L_nC_1 fault model.

7.7 Implication and Limitation

From the results, the key findings that emerge are:

1. Differences exist among different types of variables selection methods in a program in terms of their failure rates,
2. Injecting faults in TVS variables leads to higher crash rates compared to injecting into randomly selected variables,
3. Interface variables are a subset of TVS variables, and injecting into them leads to higher SDC rates,

4. Difference exist among different types of soft-errors in a program in term of their failure rates,
5. Flipping multiple bits in a single location leads to higher SDC rates as compared to flipping single bits in multiple locations, and SDC rate increases with increase in number of bits flipped,
6. Flipping single bits in a multiple location leads to higher crash rates as compared to flipping multiple bits in a single location.

The results imply that systematic selection of variables for soft-error injections potentially selects those locations that uncover more severe vulnerabilities in program as compared to random variable selection. Another implication of the results is it does not matter for the SDC rate of the program whether injections are made in full TVS variables set or in just the interface variables subset. This indicates that if the primary focus is on SDCs, then injections into interface variables subset may be sufficient for analysing the resilience of the program, compared to injections in the full TVS variables set. Similarly, the results show adopting the L_1C_m fault models may be sufficient for analysing the resilience of the program to SDCs, compared to adopting L_nC_1 fault models.

The interface variables demonstrated a completely different failure profile to that of its superset, the TVS variables. This suggests that different subset of the target variable may potentially induce different failure profile. The approach presented here is not readily applicable to neither black-box software as data dependency cannot be obtained nor to software that cannot be modelled as a control-flow graph. For such software, new techniques are required. The target systems adopted, the input sets uses and fault models assumed may have influenced the results shown, i.e., adopting different programs or input set or fault model may produce a divergent result.

7.8 Summary and Conclusions

This chapter presents heuristics to systematically select the location aspect of the L_nC_m, fault model. This framework selects target variables for injecting MBF in a single software execution. To determine these variables, the problem is split into two: (i) injection location selection and (ii) target variables selection at the possible locations. The thesis proved, in Chapter 5, both problems to be NP-complete and provided, in this chapter, two heuristics, one for each problem. Case studies have been developed to show the viability of the proposed methodology. To demonstrate the framework the chapter have shown detailed results from injecting seven versions of L_nC_m errors into ten embedded control systems.

In comparison with injection in randomly selected variables, the results show that L_nC_m uncovers more vulnerabilities. Also, the results show that the framework always include interface variables in the set of target variables suitable for injection. Injecting into interface variables uncovers more SDCs than random injections. Injecting into full set of the selected variables uncovers more severe vulnerabilities than random injections and into its interface variables subset. Injections into variables selected randomly uncovers more exception failures than injections into variables at either the interface variables subsets or the full set of variables selected by the approach.

Secondly, the results presented demonstrate that multiple errors in a single locations (L_nC_m fault models) cause higher SDC rates as opposed to single errors in multiple locations (L_nC_1 fault models), and SDC rate increases as number corruptions introduced increases for L_1C_m faults, whereas single errors in multiple locations cause more crashes and lower no impact rates as compared to multiple errors in a single locations. Thirdly, more severe vulnerabilities are uncovered concomitantly with increase in number of corruptions.

Finally, the results show that the fault space for multiple bit-flips fault injections can be effectively and systematically reduced. The next chapter will explore means of further reducing the fault space using data mining approach to discern the bit-position combinations that would induce wider failure profile.

CHAPTER 8

Learning Bits Patterns

Thus far, the thesis has focused on determining efficient target locations for the L_nC_m fault model. To this end, in the previous chapter a framework was proposed to facilitate the selection of efficient program variables that should be targeted during multiple soft-errors fault injections. This framework has facilitated in the reduction of the exponential fault space (in terms of target variables) for multiple fault injections. However, the fault injection point space (in terms of injection points) remains enormous for the L_nC_m fault model. For any given L_nC_m fault model, there are $\binom{x}{n}$ variable combinations possible, where x is the total number of all variables in the target variable set and n, the maximum number of locations to corrupt in a given execution. And, number of possible fault injections for a given location is $\binom{y}{m}$, where y is the size of the given location (in terms of the length of bits) and m is the maximum number of corruption to introduce in the said location. Thus, the fault space for the L_nC_m fault model is $\binom{x}{n} \cdot \binom{y}{m}^n$.For example, adopting a L_3C_4 fault model over six target variables, with each variable being 8-bits long, makes the number of possible injections $\binom{6}{3} \cdot \binom{8}{4}^3$, i.e., there are $1,410,760,000$ possible injection points.

This chapter attempts to fine tune the L_nC_m fault model by narrowing down the fault injection points, in terms of combination of variables to inject into and the bit-positions to perturb within a given variables combination. To this end, the chapter develops a systematic approach for the identification of efficient injec-

tion points for real world, embedded software. More specifically, the proposed approach employs data mining techniques, including decision tree induction and rule induction, for the analysis of fault injection data sets, in order to discover efficient injection points. The results presented demonstrate that this approach can be used to efficiently reduce the fault injection point space for the $\mathrm{L}_n\mathrm{C}_m$ fault model.

8.1 Data Mining in Software Dependability

Data mining techniques have been applied to address a number of other software dependability issues. In the context of improving software reliability, the application of data mining techniques have generally focused on the analysis of failure data and service logs for dependable software systems. For example, Some research [99, 101] have applied data mining techniques to improve software bugs detection. For example, Livshits and Zimmermann [99] emplodyed data mining techniques for learning common usage patterns from the revision histories of large software systems. The research analyses source code check-ins to find highly correlated method calls as well as common bug fixes in order to automatically discover application-specific coding patterns. Lo et al. [101] first mines a set of discriminative features capturing repetitive series of events from program execution traces. It then performs feature selection to select the best features for classification. These features are then used to train a classifier to detect failures. There are studies that investigates data mining techniques for the purposes of derivation of error detection predicates. For example, Pintér et al. [122] used a combination of data mining techniques on data recorded during benchmarking to identify key infrastructural factors in determining the behaviour of systems in the presence of faults. These analyses can also serve to help to identify weaknesses or vulnerabilities in a software system. Leeke et al. [91] applied data mining techniques on fault injection data to discover efficient

predicates for error detection mechanism in order to enhance dependability and address vulnerabilities in software systems.

In contrast, the data mining-based approach proposed in this chapter seeks to discover injection points for multiple soft-errors in order to enhance dependability validation and address vulnerabilities in software systems.

8.2 Data Mining Concepts

Technological advancement have resulted in generating and recording flood of data, and the amount of data information in the world is constantly rising as technology advances. These data are of no use until they are converted into useful information. Thus, it is necessary to analyse and understand this huge amount of data and extract useful information from it. The ability to extract useful knowledge hidden in these data and to act on that knowledge is becoming vital in today's increasingly information-driven world. In the case of the research presented in this chapter, it is important to understand behavioural patterns of software systems that can be used for building high-level soft-error fault models.

8.2.1 Fundamentals of Data Mining

Data mining is a technology that automatically sifts through huge amount of raw data, seeking regularities and patterns that exist therein, with the aim of using the information obtained to forecast behaviours of future data or to derive knowledge about the data, if the data itself is obscure. Data mining is well-motivated in areas where processes generate vast volume of raw data and there exist high complexity in analysing the data.

Data mining sorts through large-scale data to discover patterns and establish relationships. Technically, data mining is the process of finding correlations or

patterns among dozens of fields in large relational databases. This process of finding correlations or patterns is called learning. The (the patterns or correlations) to be learned is called a *concept* or a target function or a model. The data input used for learning the concept is a set of *instances*. Each instance is an individual, independent example of the concept to be learned. It should be mentioned that some learning tasks makes it improbable to express the raw data as individual, independent instances and often require background knowledge to be considered as part of the input. For example, learning task involving time sequence. However, the research presented in this chapter employs simple learning schemes and the data used can be presented in the form of individual instances. Each instance is characterised by the values of attributes that measure different aspects of the instance. There are different types of attribute, although the research here deals only with numeric and *nominal* (or categorical) ones. The output produced by a learning scheme is called a *concept description* or a *target function* or a *model*. Data mining learning styles include:

- **Classification:** This involves seeking novel and informative patterns. If an existing structure is already known, data mining can be used to classify new cases into these pre-determined categories. That is, the learning scheme called a classifier, is presented with classified instances from which it is expected to learn a way of classifying previously unseen instances. Classified instances are labelled with class values, and class values for new instances are determined. In the case of the research presented in this chapter, classification algorithms are applied to fault injection instances to learn, fault injection points that may likely induce failure. In a sense, classification learning operates under supervision, as the actual outcome, i.e., the class, of each learning example is given. Thus, classification learning is sometimes called supervised learning. The target function of supervised learning is a discrete function and is also referred to as a *classifier*.
- **Association:** This seeks for insightful patterns where one event is corre-

lated with another event. This seeks association between attributes, not just ones that predict a particular class value. Association learning differ from classification in two ways: (i) they can determine the value of any attribute, not just the class, and (ii) they can determine the value of more than one attribute at a time.

- **Clustering:** This involves discovering and recognising distinct categories of facts not previously known within the data. This seeks groups of instances that naturally belong together. Clustering finds these clusters and assign the instances to them, and if need be assigns new instances to the clusters.

- **Forecasting (or prediction):** Finds patterns in the data that can lead to reasonable prediction about future probabilities and trends. This area of data mining is known as predictive analytics. It is used to predict missing or unavailable numerical data values rather than class labels. Regression Analysis is generally used for prediction. Prediction can also be used for identification of distribution trends based on available data.

- **Sequence (or path analysis):** Is concerned with finding relevant patterns between data examples where the values are delivered in a sequence, i.e., where one event leads to another later event. The input data is a set of sequences called data sequences. Each data sequence is a list of transactions, where each transaction is a sets of items. A sequential pattern also consists of a list of sets of items. Sequence pattern analysis aims to find all sequential patterns with a user specified minimum support, where the support of a sequential pattern is the percentage of data sequences that contain the pattern.

The work in this chapter focuses on classification learning, hereafter, discussions are focused on concepts relating to supervised learning. In a simple domain, each instance is characterised by a set of n-attributes, the set of instances is a subset

of an n-dimensional space called an *Instance Space*, I. Every point in I is a potential state of the process being modelled. In supervised learning, a data mining algorithm is tasked with learning a good approximation, $\hat{f}$, of the target function, given a set of instances called *training data set*, T_{train}, ($T_{train} \subset I$), consisting of N pairs $< x_i, f(x_i) >$. The success of supervised learning is judged by trying out $\hat{f}$ on an independent set of instances called *test data set*, T_{test}, ($T_{test} \subset I$), for which the true classifications are known but not made known to the learner. The success rate on the T_{test} gives an objective measure of how efficiently the concept has been learned.

While there are many classification algorithms, most use the same workflow for approximating a function. Data mining involves a sequence of important steps. The steps for supervised learning include:

- **Preparing data:** This step include transforming the data into appropriate data mining format, i.e., creating data mining data sets, cleaning data and dealing with missing values, scaling and normalising data, transforming and reducing variables, partitioning data set into training, validation and test data sets, addressing class imbalance, and carrying out exploratory data analysis using graphical and statistical techniques.
- **Choosing an algorithm:** Classification algorithms include regression, decision-trees, rules induction, support vector machines (SVM), neural networks, genetic algorithms, naïve Bayes and nearest neighbours methods. The key difference between classification algorithms is in the kind of decision boundary that is defined between classes, i.e., their functional form and the set of parameters they fit, and the heuristic they employ in searching for the optimal function, also known as the hypothesis, within the space of possible hypotheses as defined by the functional form of the hypotheses.
- **Fitting a model**: This step is the learning step or the learning phase.

This involves adjusting learning parameters of the chosen classification scheme and building the model from a training data set. The learning parameters and the type of model generated are dependent on the algorithm used.

- **Choosing a validation method:** This involves selecting measures to examine the accuracy of the resulting fitted model. The model validation is done, in order to obtain a measure of its expected accuracy on unseen data. Often the accuracy of a model is evaluated with respect to the percentage of test data instances correctly classified, hence most algorithms seek to learn hypotheses that minimise the number of errors.

- **Examining fit and updating until satisfied (model refinement):** After validating the model, there might be need to change it for better accuracy, better speed, or to use less memory.

- **Using fitted model for predictions:** This involves interpreting the model and drawing conclusions.

The approach proposed in this chapter is to generate simple model to guide in the selection of efficient fault injection points. The main goal of the approach is is to detect the combination of multiple bit-flips that may likely result in system failure. For example, the approach aims to be able to predict that flipping bit 6 in variable A, bit 18 in variable B and bits 11 and 29 is variable C will most probably induce a system failure. Considering the objective of the approach and the nature of the datasets, this chapter focuses on discriminant methods to predict bit-flip combinations that may potentially cause the system to failure, as such, decision trees, rule induction and naïve Bayes algorithms are considered. The fault injection point efficiency is determined by evaluating the quality of the model produced by the learning schemes. This is done by measuring the prediction capabilities of the model.

8.3 Assessment Metrics for Model Quality

The validation methods often employed to measure the accuracy of the approximation function, implicitly assumes that all types of misclassification incur an equal cost, this however, is not always the case. For example, considering a model for a safety-critical software system which predicts either a system failure-inducing or non-failure-inducing state. Predicting a failure-inducing state as being non-failure-inducing, will typically result in a much more significant cost than classifying a non-failure-inducing state as being failure-inducing. In such situations, the predictions of a model on a test data set can be cross-tabulated with the actual classes assigned to the instances by the target function to produce a confusion matrix. Table 8.1 shows the general form of a confusion matrix for a binary classification problem (though it can easily be extended to the case of more than two classes). Classification tasks involving two classes is known as binary classification and the corresponding function approximation is known as a binary function. Typically, in binary classification, one class is the class of interest and is referred to as the "concept". The instances belonging to this concept are referred to as positive instances or positives for short. On the contrary, all other instances not belonging to the concept are called negative instances or negatives for short. In the context of the aforementioned example, (and in most software dependability analysis that uses for binary classification), the function predicts either a system state is going to lead to a system failure or a successful execution and the positives are the instances labelled as those leading to system failure.

In Table 8.1, TP is the number of positive correctly labelled as positives by $\hat{f}$, known as true positives, whilst FN is the number of positives misclassified as negatives, known as false negatives. Further, FP is the number of negatives incorrectly classified as positives, known as false positives, whilst TN is the number of negatives correctly classified as negatives, known as true negatives.

Finally, n_{pos} and n_{neg} are the respective total number of positives and negatives in the test data and $\hat{n}_{pos}$ and $\hat{n}_{neg}$ are the total number of instances predicted as positives and negatives, respectively. In software dependability analysis, it is natural to seek out models that maximise true positives and minimise false positives, not least because these correspond closely with the concepts of accuracy and completeness. However, as a balance must be struck between these related concerns, it is appropriate to identify an aggregated measures of model quality.

Table 8.1: The general form of a confusion matrix for binary classification.

		Predicted Class		
		Positives	*Negatives*	*Marginal Sums*
Actual Class	*Positives*	TP	TN	n_{pos}
	Negatives	FP	FN	n_{neg}
	Marginal Sums	$\hat{n}_{pos}$	$\hat{n}_{neg}$	n

A variety of metrics for model assessment are derived from a confusion matrix. The most commonly used of these metrics are the True Positive Rate (TPR), also known as sensitivity or recall, and true negative rate (TNR), also known as specificity. Sensitivity measures how often a model predicts an instance to be positive, when it is actually positive and it is computed as shown in Equation 8.1. Specificity measures how often a model predicts an instance to be negative when it is actually negative and it is computed as shown in Equation 8.2.

$$TPR = sensitivity = \frac{TP}{TP + FN} \tag{8.1}$$

$$TNR = specificity = \frac{TN}{TN + FP} \tag{8.2}$$

Other commonly used model validation metric include Receiver Operating Char-

acteristic (ROC) Curve (or ROC for short) and area under the curve (AUC), F-score, precision, false positive rate (FPR), mcost, accuracy and misclassification rate.

Accuracy measures how often the model is correct overall and it is computed as shown in Equation 8.3. Misclassification Rate, measures how often the model is wrong overall and it is given as shown in Equation 8.4. Misclassification rate is also known as error rate and is equivalent to 1 minus accuracy.

$$accuracy = \frac{TP + TN}{TP + TN + FN + FP} \tag{8.3}$$

$$error\ rate = 1 - accuracy = \frac{FP + FN}{TP + TN + FN + FP} \tag{8.4}$$

FPR, measures how often a model predicts an instance to be positive, when it is actually a negative. FPR is equivalent to 1 minus specificity, and its computation is shown in Equation 8.5.

$$FPR = 1 - specificity = \frac{FP}{FP + TN} \tag{8.5}$$

ROC analysis combine the FPR and the TPR into one single metric and it is based on a 2-dimensional graph that summarises the performance of a model over possible thresholds. It is generated by plotting the TPR (y-axis) against the FPR (x-axis) as the threshold for assigning instances to a given class is varied. It shows the tradeoff between sensitivity and specificity (any increase in sensitivity will be accompanied by a decrease in specificity). The closer the curve follows the left-hand border and then the top border of the ROC

space, the more accurate the test. The closer the curve comes to the 45-degree diagonal of the ROC space, the less accurate the test. For different thresholds, the same classifier will produce multiple points on such a plot. The AUC it is obtained by joining these plotted points to (0,0) and (1,1). The AUC is a measure of expected model accuracy for the classifier. For a single model, the simple trapezium obtained by connecting the coordinates (0,0), (FPR,TPR), (1,1) and (1,0) has an area given by Equation 8.6. An area of 1 represents a perfect model.

$$AUC = \frac{TPR - FPR + 1}{2} \tag{8.6}$$

The Euclidean distance from the perfect classifier, which has coordinates (0, 1), i.e, FPR = 0 and TPR = 1, may be used in the ranking of single models. This measure is given by the well known formula in Equation 8.7.

$$euclidean\ distance = \sqrt{(FPR - 0)^2 + (1 - TPR)^2} \tag{8.7}$$

Precision measure how often a model prediction is correct when it predicts yes and is given by Equation 8.8. Precision and Recall together with their harmonic mean F_1 measure or balanced F-Score (F_1-Score) are often used as a model quality metric in the domain of information retrieval. The F-Score is a weighted average of the recall and precision, and it is measured as shown in Equation 8.9.

$$precision = \frac{TP}{TP + FP} \tag{8.8}$$

$$F_1\text{−}Score \;=\; 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{8.9}$$

When specific classification errors are more severe than others, e.g., When the cost associated with a false positive is different from that of a false negative, a more appropriate measure of quality is expected, misclassification cost, rather than the expected error. This requires the definition of a cost matrix. Assuming there are m class labels, L_i, an $m \times m$ cost matrix, C, needs to be defined such that the value $C(i,j)$ is the cost of misclassifying an instance of class L_i to the class L_j. Clearly $C(i,i) = 0$ as there should be no cost associated with correctly classifying an instance. The model tries to avoid classification errors with a high error weight. Weights specified must be greater than or equal to zero. The default weight is 1. The cost matrix diagonal must be zero. Minimising the error is a special case of minimising misclassification cost when the cost matrix is defined as $C(i,j) = 1$, where $i \neq j$ and $C(i,i) = 0$. The expected misclassification cost, $mcost$, can then be calculated as shown in Equation 8.10, where $CM(i,j)$ represents index access to the associated confusion matrix using i and j.

$$mcost \;=\; \sum_{i}^{m}\sum_{j}^{m} C(i,j) \cdot CM(i,j) \tag{8.10}$$

The proposed approach in this chapter focuses on the generation of efficient fault injection points, which means that the model quality is evaluated with respect to the efficiency, i.e., the levels of accuracy and completeness, that can be achieved by these points. With this in mind, the AUC measure, which is an aggregate representative of accuracy and completeness in the form of TPR and FPR, is used in measuring model quality. However, as misclassification

costs are likely to vary in the context of dependable software systems, steps must be taken to ensure that high AUC values are not achieved through the neglect of accuracy or completeness. Having this in mind, TPR and FPR are also considered when evaluating the quality of generated models.

8.4 Addressing Class Imbalance

Typically standard learning algorithms, in addition to assuming equal misclassification costs, expect the balanced class distributions across the training data [56, 63]. Therefore, when presented with complex imbalanced data sets, these algorithms fail to properly represent the distributive characteristics of the data and resultantly provide unfavourable accuracies across the classes of the data. To improve the performance of learning algorithms in the presence of under represented data and severe class distribution skews, there is need to balance the data efficiently. However, there are a number of domains, such as error detection, intrusion detection, fraud detection and software reliability, where the class distribution is skewed. Often in a binary classification, the positive class is the minority class and the class of interest. For example, in the context generating error detection mechanisms, detecting system failures is of most interest and it is the minority class [91].

The approach proposed here assumes that the data generated during fault injection analysis generally captures aspects of relationships between system states and systems failures, in particular it captures relationships between bit-positions and failure profile. Based on the sampled states and observed system behaviour during fault injection analysis, a data mining algorithm can then determine fault injection points through learning about these captured relationships. Typically, data sets generated from fault injection analysis are skewed, i.e., some failure mode occur less frequently than others. Such imbalance data sets must first be processed for the learning process to be effective with respect to generating

efficient fault injection points.

There are two common used approaches to overcome the effects of imbalanced data. The first of these, modifies the imbalance data set by some sort of sampling approach in order to provide a balanced distribution [63, 80, 96]. A variety of sampling methods have been proposed. Oversampling and undersampling are the most commonly sampling methods in usage. They are opposite and roughly equivalent approaches. They both involve using a bias to select more samples from one class than from another. In the case of oversampling, original data set is increased by a set of replicated instances from the minority class, and the these resampling can be with or without replacement. Undersampling involves decreasing the original set by removing some instances of the majority class, and resampling is done without replacement. These approaches introduces their own set of issues that can potentially hinder model performance. Removing instances of majority class can potentially cause the learning scheme miss important concepts regarding the majority class. On the other hand, adding replicated instances may potentially cause the learning scheme to overfit. The former is a problem related to undersampling and the latter to oversampling [110]. To circumvent around these issues and to improve model accuracy Chawla et al. [18] demonstrated the use of cross validation for setting the level of oversampling and undersampling of the majority and minority classes automatically. Some research proposed informed undersampling and oversampling methods to ameliorate these issues. Zadrozny et al. [166] suggest the use of a cost-proportionate rejection sampling technique, while Kubat and Matwin [80] proposed removing redundant and borderline negative instances during undersampling. However, Japkowicz [63] demonstrated that oversampling from the boundary regions and undersampling far from the decision boundary adds little value over random sampling approaches. Chawla et al. [17] proposed a method, Synthetic Minority Oversampling Technique (SMOTE), to generate synthetic data for minority classes along the line segment joining an instance to k minority class nearest

neighbours rather than simply sampling with replacement. They demonstrated that SMOTE outperforms simple sampling with replacement.

The second approach uses cost-sensitive learning methods to consider the costs associated with misclassifying instances. Typically, they set higher cost associated with misclassifying instances of the minority class by defining a cost matrix based on the class imbalance and then use the same error minimisation-based concept learning algorithms. However, this approach assumes that such a cost matrix can be incorporated by the learning process. There are some methods that replace error minimisation metrics with cost minimisation metrics when searching the hypothesis space. However, Pazzani et al. [120] et al. demonstrated that adopting misclassification costs as a greedy selection criteria in decision tree induction does not provide cost minimisation for the model generated. Further, Ting [157] using instance weighting is more effective than using a cost minimisation-based approach. However, assigning distinct weights to training instances, in effect, modifies the data distribution within the training data [34, 41, 120, 157]. Typically, it is necessary to convert cost matrix into cost vector, V, and this may potentially be difficult in the case of multi-class classification problems. Breiman et al. [12] proposed using the sum of all misclassification costs for instances of the class. Ting [157] uses an alternative weight, $V(i) = arg \max_j(C(i,j))$, to assign weight to all instances of a particular class, L_j, based on $V(j)$ using the formula shown in Equation 8.11, where N_j is the number of instances in the data labelled L_j and $N = \sum_i N_i$.

$$w(j) = V(j) \cdot \frac{N}{\sum_i V(i) \cdot N_i} \tag{8.11}$$

Another noteworthy approach adopted for imbalanced learning problems is active learning methods [38, 39]. Traditionally, active learning methods are used to solve problems related to unlabelled training data. The prime concept of

active learning is that a machine learning algorithm can achieve high accuracy with small amount of training labels. This is achieved by allowing the learning algorithm to interactively query an information source to provide for data labels [143]. Active learning approaches for imbalanced learning are often integrated into kernel-based learning methods. For instance, Ertekin et al. [39] uses SVM-based active learning to select instances that are closest to the current hyperplane. In [38, 39], they proposed an efficient SVM-based active learning method which queries a small pool of data at each iterative step of active learning instead of querying the entire data set. In this procedure, an SVM is trained on the given training data, after which the most informative instances are extracted and formed into a new training set according to the developed hyperplane. Finally, the procedure uses this new training set and all unseen training data to actively retrain the SVM using the LASVM online SVM learning algorithm [10] to facilitate the active learning procedure.

8.5 Generating Fault Injection Points

The approach presented in this chapter generates efficient fault injection points in three stages. The initial stage of the process is done in two steps: Step 1, involves performing fault injection analysis by perturbing identified target variables in a program with multiple soft-errors, in order to capture the relationship between these perturbations and their impact on the program. The logged data can be used for generating efficient fault injection points. Step 2 involves preprocessing the fault injection analysis data. The aim of this step is to: (i) transform the fault injection data into an appropriate format for usage in data mining analysis, and (ii) to address any class imbalance in the generated data sets. Stage two of the process consists of two steps: Step 1 requires choosing an appropriate data mining algorithm and adjusting parameters that will improve the effectiveness of the chosen algorithm. In step 2, the chosen learner is applied

to transformed and balanced data set, in order to produce and assess the first fault injection points. The third and final, stage, adjusts the fitting parameters of the learning scheme, if need be, to improve the injection efficiency in terms inducing failure. An overview of the process of generating efficient fault injection points in shown in Figure 8.1, and the process in elaborated in Sections 8.5.1-8.5.3.

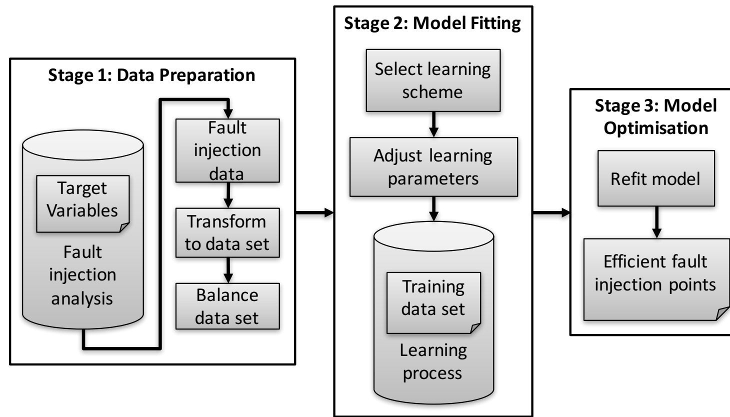


Figure 8.1: Workflow for generating efficient fault injection points.

8.5.1 Stage 1: Data Preparation

The objective of this is to generate data mining sets from fault injection analysis. Thus, the initial step of the proposed approach is to perform fault injection experimentations on a target software system in order to log aspects of the relationship between system state and failure failure, specifically bits being flipped in variables and failure mode. The precise nature of the fault injection performed is dependent on the assumed fault and system models, which in-turn is dependent on the characteristic of the target program. This means, there will be direct relationship between the nature of the fault and system models assumed and the nature of (fault injection point) model that will be derived. The importance of fault models and input sets assumed for fault injection analysis to make dependability enhancement meaningful cannot be overemphasised. As

such, the relationship of systems states to system failure not captured by the assumed fault model may not potentially be discerned in data mining process. For example, in this thesis multiple soft-errors, L_nC_m fault model (See Chapter 7.5) is assumed, which means that the set of variables and bit-flips combination not captured by the adopted fault models, may not necessarily be accounted for by the derived fault injection point model. The results presented in this chapter are based on sampling all variables in the set returned by TVS (See Chapter 7.5.1). However, additional sampling was performed for the assumed single and double faults, all bit-flips combinations was considered.

Following the analysis of the fault injection data, the generated data is converted to appropriate data data mining datasets. The motivation for this step is to: (i) transform the fault injection analysis data to a format that can be processed by classification algorithms, and (ii) any address any class distribution skewness present in the derived datasets. In the case of the results presented in this chapter, the data format transformation was from LLFI [103] logging format to the Attribute-Relation File Format (ARFF) used by the Weka Data Mining suite [54]. Often, classes have very unequal frequency, like those found within datasets obtained through fault injection analysis. This occurs as a result of the inherent resilience of software and difficulty of inducing system failures under a given fault model. To generate a reliable model that will predict effective fault injections points that might induce system failures, any such imbalance will have to be addressed. This imbalance are usually addressed through sampling approaches such as undersampling of the majority class and oversampling with replacement of the minority class. However, with multiple fault injections analysis the imbalance for a binary valued class is minimal and may likely yield reliable models even if no sampling is done to balance the dataset. Oversampling can be viewed as a case of SMOTE [17]. In SMOTE, synthetic instances of the minority class are generated by operating in the "feature space" rather than the "data space". SMOTE creates new instances by taking samples of the

feature space of the minority class, and its nearest neighbours, and generates new examples that combine the features of the target case with features if its neighbours. This approach increases the features available to each class and makes the samples more general. The synthetic are generated as defined in Equation 8.12.

$$\overrightarrow{s}_{ij} = \overrightarrow{x_{ij}} + u \cdot (\overrightarrow{n_{ij}} - \overrightarrow{x_{ij}}) \tag{8.12}$$

with, $\overrightarrow{x_{ij}}$ representing the instances of the minority class, $\overrightarrow{n_{ij}}$ representing $\overrightarrow{x_{ij}}$'s neighbours (randomly sampled with replacement from k of its nearest neighbours), and u is a random number, such that $0 \leqslant u \leqslant 1$. Oversampling with replacement is a case of SMOTE where $u = 0$.

However, the task of addressing class imbalance can not completed until data mining has been used to fit some initial model, hence it is an aim that is only realised during the optimisation of the generated model, as described in Section 8.5.3.

8.5.2 Stage 2: Model Fitting

The aim of the this stage of the approach is to generate first-attempt fault injection point model from the transformed fault injection data. Following the generation of the datasets, an appropriate data mining algorithm must be selected for data analysis. To derive a discriminant model to predict combinations of multiple bit-flips over program variables that may potentially induce system failure. The use algorithms that generate intuitive models and are easily interpreted is advocated.

The next point after choosing the learning algorithm is to adjust the parameters of the algorithm to improve the model's performance and start the learning process. At this point the aim is not necessarily to generate highly-efficient fault

injection point model, but to establish a baseline model that can be optimised and refined in the next stage of the approach. The validation of the generated model may take place as flipping the relevant bits at the relevant program location in the target program and observe the outcome of the program execution or by testing the efficacy of the model's prediction of unseen instances, i.e, instances not used in generating the fault injection point model. The purpose, in either case, is to assess the quality of the model's prediction on previously unseen data in order to measure efficiency properties.

8.5.3 Stage 3: Model Optimisation

Once a baseline model has been generated and evaluated, it may be refined in order to improve its level of coverage. This can be achieved by varying the parameters associated with the configuration of the adopted learning algorithm. If class imbalance is present in the data set, it is useful to vary the levels of undersampling or oversampling, including the number of nearest neighbours used by any sampling techniques applied, in order to establish an algorithm configuration which yields the most efficient fault injection points. An ideal and optimal model will have $TPR = 1$, $FPR = 0$ and $AUC = 1$. In reality, it may be infeasible to achieve an ideal model since the sample is severely constricted. However, the aim of the approach is to achieve measures as close to the ideal model as possible.

8.6 Case Studies

To demonstrate that the proposed approach for the generation of fault injection point yields efficient fault injection points prediction model, the results of applying each stage of the approach are presented in Sections 8.6.1-8.6.3.

8.6.1 Stage 1: Data Preparation

In order to generate datasets, fault injection analysis was conducted on all target systems under the experimental conditions described in Chapters 3 and 7. During fault injection it is possible to inject specific bit or bits combinations in the specific program location or combination of locations and then log the execution outcome. Broadly, the program location at which an injection is performed and what is injected will dictate the set of erroneous states explored. The variable locations were chosen using the framework in Figure 7.2. The results of fault injection analysis were stored in the LLFI analysis and logging format [103]. A purpose-built was used to convert from the LLFI analysis and logging format to the ARFF format used by the Weka Data Mining Suite [54]. The datasets generated in this chapter are multiple class datasets. The failure scheme adopted for labelling the instances is as defined in Chapter 3.4.3, however No Impact outcomes are classed as non-failures in the generated datasets. As mention earlier in this chapter, class imbalance may not be present in a multiple fault injection data, and this proved to be true in the case of the analysis used in this chapter. In some instances the minority class, are the negatives instances, i.e the non-failure class. In the case of the results presented in this chapter, the failure classes, i.e., crash, hang, sdc and exception, are considered to be the positives. 30 datasets were generated, for ten target programs. For each target program, three datasets were generated: (i) capturing the aspect of the relationship of the system behaviour to injections of not more than two bit-flips (double-bits datasets), (ii) capturing the aspect of the relationship of the system behaviour to injections of not more than three bit-flips (triple-bits datasets), and (iii) capturing the aspect of the relationship of the system behaviour to injections of not more than four bit-flips (quadruple-bits datasets). This means, that the double-bits datasets capture system behaviour related to single-bit-flips and double-bit-flips; the triple-bits datasets are double-bits datasets extended to also capture system behaviour relevant to triple-bit-flips; and the quadruple-

bits datasets are extended triple-bits datasets capturing additional aspect of system behaviour relevant to quadruple-bit-flips. An example of ARFF format, consisting up to two double bits faults (one or two bit-flips injected injected in one or two program locations) and five experiments, is depicted in Figure 8.2.

```
@RELATION isqrtDouble

@ATTRIBUTE loc1 {var1,var2,var3,var4,var5,var6,var7,none}
@ATTRIBUTE loc2 {var1,var2,var3,var4,var5,var6,var7,none}
@ATTRIBUTE pos1 NUMERIC
@ATTRIBUTE pos2 NUMERIC
@ATTRIBUTE failure {no-impact,sdc,exception,time-out,crash}

@DATA
var1, 1, var1, 7, no-impact
var1, 2, var2, 5, sdc
var4, 23, none, 0, crash
var3, 8, var3, 9, sdc
var6, 2, var7, 51, sdc
```

Figure 8.2: An overview of generated data set.

8.6.2 Stage 2: Model Fitting

Following the generation of the fault injection datasets and their pre-processing, the following classification algorithms are selected to build the fault injection points predictive models:

Naïve Bayes: The naïve Bayes classification algorithm is a probabilistic classifier based on applying Bayes' theorem. It assumes that the value of a particular feature is unrelated to the presence or absence of any other feature, given the class variable [71]. Despite its apparent over-simplified assumption, the naïve Bayes classifier have worked quite-well in many complex real-world applications. The naïve Bayes classifier estimates the prior probability distribution of the classes, i.e., crash, sdc, exception, hang and non-failure in the case of learning error fault injection points, and the class conditional probabilities of input vectors. It assumes conditional independence of input variables given the class. Given an input vector, x, it assigns the class label that has the maximal

posterior probability, as defined in Equation 8.13.

$$c_i = arg \max_{c_i} p(c_i|x) = arg \max_{c_i} \frac{\prod_{j=1}^{n} p(c_i|x)p(c_i)}{\sum_k \prod_{j=1}^{n} p(c_k|x)p(c_k)} \tag{8.13}$$

In the case of continuous input attributes, kernel density estimation is used to estimate the class conditional probability density functions as opposed to the common assumption of a single Gaussian distribution. The implementation of the naïve Bayes classification algorithm used to generate the results presented in this chapter employs the gaussian kernel, g, as shown in 8.14

$$p(X_i = x|c_j) = \frac{1}{n} \sum_k g(x; x_k; \sigma_j) \tag{8.14}$$

Rule Induction: Rule induction is a desirable approach to learning because the knowledge generated is a set of conjunctive rules that are easy to understand. Repeated Incremental Pruning to Produce Error Reduction (RIPPER) is a propositional rule inducer and it is used as the rule induction implementation for the results presented in this chapter [22]. Rule inducers are algorithms that iteratively generates a rule that covers a subset of the training data, and removing all the examples covered by the rule from the training data until there are no more examples to cover. RIPPER builds a ruleset by repeatedly adding rules to an empty ruleset until all positive examples are covered. Starting with an empty antecedent, rules are formed by greedily adding conditions to the antecedent of a rule until no negative examples are covered.

Decision Trees Induction: Decision tree inducers are algorithms that automatically construct a decision tree from a given dataset. Typically the goal is to find the optimal decision tree by minimising the generalisation error. However, other target functions can be also defined, for example, minimising the number

of nodes or minimising the average depth. The decision tree induction algorithm implementation used to generate the results in this chapter is the C4.5 [128]. The C4.5 builds decision trees from a set of training data, using the concept of information entropy. At each node of the tree, C4.5 chooses the attribute of the data that most effectively splits its set of examples into subsets enriched in one class or the other. The splitting criterion is the normalised information gain (difference in entropy). The attribute with the highest normalised information gain is chosen to make the decision. The C4.5 algorithm then recurs on the smaller sublists.

Following the selection of the three classification algorithms, 10-fold cross validation was used in order to generate the confusion matrix for each algorithm on each data set. In 10-fold cross validation the entries in each data set are partitioned into 10 stratified samples, then for each cross validation run, one of these partitions is used as a test sample, whilst the other nine are used as the training set for a particular classification algorithm.

Tables 8.2-8.4, Tables 8.5-8.7 and Tables 8.8-8.10 summarise the results of applying the naïve Bayes, rule induction and decision tree induction data mining algorithms respectively, to each fault injection data set. The statistics shown in these tables relate to fault injection points predictive models generated using a baseline configuration of each data mining algorithm, i.e., no attempt was made to search for algorithm parameters that would yield the most effective predictive models. In these table, the TPR and FPR columns give the mean true positive and true false rates taken across all 10 cross validations. A false positive here corresponds to the situation where a model incorrectly detects a state as being failure-inducing (i.e., either as crash, hang, sdc or exception), whilst a true positive corresponds to a model correctly identifying a failure-inducing state. The AUC column shows the area under the ROC curve, as described in Section 8.3, whilst the SD column gives the standard deviation in AUC across all 10 cross validations. Before proceeding with analyses of the results, the key points are

summarised as follows:

1. Baseline naïve Bayes may not always produce efficient fault injection points predictive models,
2. Baseline decision tree induction algorithm produces efficient fault injection points predictive models, and it out performs both the rule induction and naïve Bayes classifiers,
3. The baseline classifiers tend to generate more efficient fault injection points predictive models from datasets derived from fault injection analysis with higher number of corruptions.

Table 8.2: (Double) Injection points efficiencies for naïve Bayes with no sampling

Data Set	**TPR**	**FPR**	**AUC**	**SD**
crc	0.86023	0.03119	0.91452	0.02739
dijkstra	0.93890	0.09997	0.91946	0.01848
fft	0.90499	0.01005	0.94747	0.03495
search	0.95107	0.11872	0.91618	0.01436
insert	0.86747	0.04987	0.90880	0.01194
remove	0.86932	0.07048	0.89943	0.03901
encfile	0.86353	0.05394	0.90480	0.03649
decfile	0.90345	0.11794	0.89276	0.03851
isqrt	0.89714	0.04432	0.92641	0.03177
cubic	0.89510	0.06630	0.91440	0.02810

The results shown in Table 8.2 relate to fault injection points predictive models generated by the naïve Bayes algorithm for the double-bits datasets. It can be observed that the predictive models generated for each dataset have varied TPR values, with entries in Table 8.2 being in the range 0.86023 to 0.95107. The value of mean FPR for naïve Bayes are equally diverse across different datasets, with these values being in the range 0.01005 to 0.11872. In general, the TPR and FPR values shown in Table 8.2 mean that the worst performing predictive model generated by naïve Bayes may not have the levels of efficiency that are required in the context of dependable software. In contrast, the best performing of these predictive models may be useful in the design of dependable

software. For example, the predictive model associated with *fft* have a TRP and FRP of 0.90499 and 0.01005 respectively, yielding a promising AUC of 0.94747. Perhaps the most interesting characteristic of the results presented in Table 8.2 is the consistently low standard deviation in mean AUC, which indicates that high levels of injection efficiency, i.e., TPR and FPR rates, were consistently achieved during each of the 10 cross validations.

Table 8.3: (Triple) Injection points efficiencies for naïve Bayes with no sampling

Data Set	**TPR**	**FPR**	**AUC**	**SD**
crc	0.86884	0.03088	0.92367	0.02767
dijkstra	0.94829	0.09897	0.92866	0.01866
fft	0.91404	0.00995	0.95694	0.03530
search	0.96058	0.11753	0.92534	0.01450
insert	0.87614	0.04937	0.91789	0.01206
remove	0.87802	0.06977	0.90842	0.03940
encfile	0.87216	0.05340	0.91384	0.03685
decfile	0.91249	0.11676	0.90168	0.03889
isqrt	0.90611	0.04388	0.93567	0.03209
cubic	0.90405	0.06564	0.92354	0.02838

The results presented in Table 8.3 demonstrate that the fault injection point predictive models generated by the naïve Bayes classifier for the triple-bits datasets are comparable with, but marginally more efficient than, those generated generated for double bits, with all mean AUC values being in the range 0.86884 to 0.96058. Indeed, the predictive models generated from triple bits classifier surpassed the classifier trained with double bits, with respect to mean AUC. Interestingly, the standard deviation in mean AUC remains consistently low, again indicating the consistency with which similarly efficient fault injection point predictive models are generated during cross validation.

The results presented in Table 8.4 indicate that the fault injection point predictive models generated using the naïve Bayes algorithm for the quadruple-bits datasets marginally surpass those generated for both double-bits and triple-bits with respect to the level of efficiency achieved, with all mean AUC values in Table 8.3 being in the range 0.91070 to 0.96651. The standard deviation in

Table 8.4: (Quadruple) Injection points efficiencies for naïve Bayes with no sampling

Data Set	TPR	FPR	AUC	SD
crc	0.87752	0.03057	0.93290	0.02794
dijkstra	0.95777	0.09798	0.93794	0.01885
fft	0.92318	0.00985	0.96651	0.03565
search	0.97019	0.11636	0.93459	0.01465
insert	0.88490	0.04888	0.92707	0.01218
remove	0.88680	0.06907	0.91751	0.03979
encfile	0.88088	0.05286	0.92298	0.03722
decfile	0.92161	0.11559	0.91070	0.03928
isqrt	0.91517	0.04344	0.94503	0.03241
cubic	0.91309	0.06498	0.93278	0.02866

AUC is also lower than for the predictive models built with either double-bits or triple-bits datasets, with the highest observed standard deviation being less than the lowest value associated with with both double-bits and triple-bits . In general, the results associated with naïve Bayes may not have the levels of efficiency with respect to the generation of efficient fault injection points predictive models, that being said, optimising the model may boost the levels of efficiency to acceptable levels.

Table 8.5: (Double) Injection points efficiencies for rule induction with no sampling

Data Set	TPR	FPR	AUC	SD
crc	0.94828	0.01933	0.96448	0.00011
dijkstra	0.93003	0.03147	0.94928	0.00217
fft	0.92317	0.07812	0.92252	0.00026
search	0.95087	0.05051	0.95018	0.00039
insert	0.97817	0.03475	0.97171	0.00116
remove	0.96485	0.03929	0.96611	0.00045
encfile	0.93927	0.01716	0.96106	0.00564
decfile	0.97961	0.05570	0.96195	0.00017
isqrt	0.92964	0.04583	0.94190	0.00017
cubic	0.94930	0.04140	0.95440	0.00120

The results presented in Table 8.5, Table 8.5 and Table 8.5 indicate that the fault injection point predictive models generated by the decision rule inductor for the double-bits, triple-bits and quadruple-bits datasets surpasses those generated

under naïve Bayes, with all mean AUC values being in the range 0.92252 to 0.97171, 0.93175 to 0.98143 and 0.94107 to 0.99124 respectively. The standard deviation in AUC is also markedly lower than for all the naïve Bayes classifiers.

Table 8.6: (Triple) Injection points efficiencies for rule induction with no sampling

Data Set	**TPR**	**FPR**	**AUC**	**SD**
crc	0.95776	0.01913	0.97412	0.00011
dijkstra	0.93933	0.03116	0.95877	0.00219
fft	0.93241	0.07734	0.93175	0.00026
search	0.96038	0.05001	0.95968	0.00040
insert	0.98795	0.03441	0.98143	0.00117
remove	0.97450	0.03889	0.97577	0.00045
encfile	0.94866	0.01699	0.97067	0.00570
decfile	0.98940	0.05515	0.97157	0.00017
isqrt	0.93894	0.04537	0.95132	0.00017
cubic	0.95879	0.04099	0.96394	0.00121

Similar to the trend observed under naïve Bayes, the rule inducers models from the triple-bits datasets outperformed those model from double-bits datasets, and those trained on quadruple-bits surpassed those generated from triple-bits datasets. The worst results observed under the rule inducers was in the double-bits dataset for *fft*, having a TPR and FPR of 0.92317 and 0.07812.

Table 8.7: (Quadruple) Injection points efficiencies for rule induction with no sampling

Data Set	**TPR**	**FPR**	**AUC**	**SD**
crc	0.96734	0.01894	0.98386	0.00011
dijkstra	0.94873	0.03085	0.96836	0.00221
fft	0.94173	0.07657	0.94107	0.00026
search	0.96998	0.04951	0.96928	0.00040
insert	0.99783	0.03406	0.99124	0.00118
remove	0.98424	0.03850	0.98553	0.00046
encfile	0.95815	0.01682	0.98038	0.00576
decfile	0.99930	0.05459	0.98129	0.00017
isqrt	0.94833	0.04491	0.96084	0.00017
cubic	0.96838	0.04058	0.97358	0.00122

In general, the results associated with rule induction are promising with respect to the generation of fault injection point predictive models, not least because

these results relate to a baseline configuration of the rule induction algorithm.

Table 8.8: (Double) Injection points efficiencies for decision tree induction with no sampling

Data Set	TPR	FPR	AUC	SD
crc	0.95933	0.00004	0.97965	0.00579
dijkstra	0.99011	0.00143	0.99434	0.00017
fft	0.99817	0.00001	0.99808	0.00012
search	0.85785	0.00009	0.92888	0.00062
insert	0.98649	0.00010	0.99320	0.00009
remove	0.98638	0.00133	0.99253	0.00124
encfile	0.97999	0.00104	0.98947	0.00011
decfile	0.97363	0.00000	0.98681	0.00000
isqrt	0.99551	0.00011	0.99770	0.00015
cubic	0.96970	0.00050	0.98460	0.00090

Table 8.5, Table 8.7 and Table 8.7 suggest that decision tree induction is the most effective of the classification algorithms applied to this point. Observe from Tables 8.5-8.7 that the mean AUC of all baseline predictive models generated through decision tree induction is greater than 0.97965. As this measure reflects both FPR and TPR, this is an indication that the injection points generated are effective classifiers for failure inducing injection points. Observe also that, aside from datasets *search*, the mean TPR for all decision trees is greater than 0.95933 (using the double-bits datasets), with the maximum observed being 0.99937 (using the quadruple-bits datasets). Further, the mean FPR is extremely low in all cases, with the maximum observed value being 0.00143 (in all the dijkstra datasets).

As with naïve Bayes and rule induction classifiers, the predictive models efficiency derived from the quadruple-bits surpasses those generated from the triple-bits, and in-turn, the classifiers models from the triple-bits datasets outperformed those derived from the double-bits datasets.

This indicates the discriminatory nature of the injection points generated by the decision tree induction algorithm. It is also interesting to note that the standard deviation of the injection points generated, regardless of the data mining

Table 8.9: (Triple) Injection points efficiencies for decision tree induction with no sampling

Data Set	TPR	FPR	AUC	SD
crc	0.96029	0.00004	0.98063	0.00580
dijkstra	0.99110	0.00143	0.99534	0.00017
fft	0.99917	0.00001	0.99908	0.00012
search	0.85871	0.00009	0.92981	0.00062
insert	0.98748	0.00010	0.99419	0.00009
remove	0.98737	0.00133	0.99352	0.00124
encfile	0.98097	0.00104	0.99046	0.00011
decfile	0.97460	0.00000	0.98780	0.00000
isqrt	0.99650	0.00011	0.99870	0.00015
cubic	0.97067	0.00050	0.98558	0.00090

Table 8.10: (Quadruple) Injection points efficiencies for decision tree induction with no sampling

Data Set	TPR	FPR	AUC	SD
crc	0.96048	0.00004	0.98083	0.00580
dijkstra	0.99130	0.00143	0.99554	0.00017
fft	0.99937	0.00001	0.99928	0.00012
search	0.85888	0.00009	0.93000	0.00062
insert	0.98768	0.00010	0.99439	0.00009
remove	0.98756	0.00133	0.99372	0.00124
encfile	0.98116	0.00104	0.99066	0.00011
decfile	0.97480	0.00000	0.98800	0.00000
isqrt	0.99670	0.00011	0.99890	0.00015
cubic	0.97086	0.00050	0.98578	0.00090

algorithm applied, is consistently low, which demonstrates the consistency with which efficient injection point predictive models can be generated when using a decision tree induction-based approach.

8.6.3 Stage 3: Optimising Model

Following the fitting of a set of baseline fault injection points predictive models and validating their performance, these predictive models can now be refined by varying the parameters associated with the applied classification algorithms. In particular, it is interesting to vary parameters that are independent of any algorithm, such as dataset sampling levels prior to learning. This permit the

same optimsation process to be applied regardless of the selected classification algorithm. Tables 8.11-8.19 summarises the results of the model optimisation process for the presented case studies. Tables 8.11-8.19 are similiar to those given in Tables 8.2-8.10, except that Tables 8.11-8.19 have addition columns, Sampling and N, which show the level of sampling and the nearest neighbours used in sampling to generate the associated predictive models respectively. Each entry in the Sampling column also shows the type of sampling performed, where an O denotes oversampling and a U, undersampling. A total of 20 undersampling and 15 oversampling percentage levels were used in model refinement. These levels were uniformly distributed over $[5, 100]$ and $[100, 1500]$ for undersampling and oversampling respectively, giving increments of 5 and 100 respectively. The number of nearest neighbours considered in the sampling process were uniformly distributed over $[1, 15]$ with increments of 1. The values in the Sampling and N columns of Tables 8.11-8.19 represent optimal observed values, with regard to achieved AUC, across all candidate values considered.

Before proceeding with analyses of the results, the key points are summarised as follows:

1. Optimising the classifier can improve the efficiency of the fault injection points predictive models,
2. Similar with the baseline classifiers, optimised decision tree induction outperforms both naïve Bayes and rule induction classifiers,
3. Similar with the baseline classiffiers, the optimised classifiers tend to produce more efficient fault injection points predictive models on datasets derived from fault injection analysis with higher number of corruptions.

The results presented in Tables 8.11-8.13 improve on the results presented for the naïve Bayes classifiers in Tables 8.2-8.4, clearly demonstrating that varying the sampling parameters associated with the application of naïve Bayes can

improve the efficiency of the fault injection points predictive models. More specifically, all mean TPR and FPR values have been improved, which lead to a increase in mean AUC. The standard deviation in AUC is consistently low and remains comparable with the results generated under a baseline configuration of the naïve Bayes classifiers.

Table 8.11: (Double) Injection points efficiencies for naïve Bayes with sampling

Data Set	**Sampling**	**N**	**TPR**	**FPR**	**AUC**	**SD**
crc	500 (O)	8	0.90831	0.00363	0.95234	0.05592
dijkstra	1000 (U)	4	0.95237	0.08128	0.93555	0.03027
fft	300 (O)	3	0.93064	0.00211	0.96426	0.17206
search	200 (O)	2	0.97193	0.05788	0.95703	0.05469
insert	500 (O)	8	0.91706	0.01364	0.95171	0.04190
encfile	600 (O)	8	0.89438	0.02542	0.93448	0.05305
decfile	600 (O)	6	0.94737	0.05728	0.94505	0.06371
isqrt	600 (O)	0	0.92111	0.00476	0.95817	0.08239
cubic	500 (O)	0	0.92380	0.03140	0.94620	0.06770

Table 8.12: (Triple) Injection points efficiencies for naïve Bayes with sampling

Data Set	**Sampling**	**N**	**TPR**	**FPR**	**AUC**	**SD**
crc	500 (O)	8	0.90922	0.00362	0.95330	0.05598
dijkstra	100 (O)	4	0.95332	0.08120	0.93648	0.03030
fft	300 (O)	3	0.93157	0.00211	0.96523	0.17223
search	200 (O)	2	0.97290	0.05782	0.95798	0.05474
insert	500 (O)	5	0.90637	0.05811	0.92457	0.05174
remove	400 (O)	8	0.91798	0.01363	0.95266	0.04195
encfile	600 (O)	8	0.89527	0.02539	0.93542	0.05311
decfile	600 (O)	6	0.94832	0.05722	0.94600	0.06378
isqrt	700 (O)	0	0.92203	0.00476	0.95913	0.08247
cubic	600 (O)	0	0.92472	0.03137	0.94715	0.06777

All entries in Tables 8.14-8.16 indicate that the optimisation process has improved the efficiency properties of the fault injection points predictive models derived with rule induction. Indeed, the results show an improved mean AUC across all entries. The standard deviation in mean AUC is easily comparable with standard deviation observed under a baseline configuration of rule induction, with some generated predictive models even yielding a reduction in standard deviation with respect to mean AUC.

Table 8.13: (Quadruple) Injection points efficiencies for naïve Bayes with sampling

Data Set	Sampling	N	TPR	FPR	AUC	SD
crc	500 (O)	8	0.91013	0.00362	0.95425	0.05603
dijkstra	100 (O)	4	0.95427	0.08112	0.93742	0.03033
fft	300 (O)	3	0.93250	0.00211	0.96619	0.17240
search	200 (O)	2	0.97387	0.05776	0.95894	0.05480
insert	500 (O)	5	0.90728	0.05805	0.92550	0.05179
remove	400 (O)	8	0.91890	0.01362	0.95361	0.04199
encfile	600 (O)	8	0.89617	0.02537	0.93635	0.05316
decfile	600 (O)	6	0.94927	0.05716	0.94694	0.06384
isqrt	700 (O)	0	0.92295	0.00475	0.96009	0.08255
cubic	600 (O)	0	0.92565	0.03134	0.94809	0.06784

Table 8.14: (Double) Injection points efficiencies for rule induction with sampling

Data Set	Sampling	N	TPR	FPR	AUC	SD
crc	700 (O)	2	0.96822	0.01933	0.97445	0.01780
dijkstra	500 (U)	6	0.98868	0.00889	0.98990	0.00910
fft	300 (O)	6	0.92984	0.05797	0.93594	0.00160
search	400 (O)	8	0.95420	0.03709	0.95855	0.04416
insert	700 (O)	3	0.96508	0.00473	0.98017	0.01165
remove	600 (O)	3	0.98460	0.01028	0.98716	0.02927
encfile	500 (O)	3	0.96973	0.00030	0.98472	0.00416
decfile	400 (O)	2	0.98961	0.05460	0.96751	0.00160
isqrt	400 (U)	9	0.92964	0.04583	0.94190	0.01028
cubic	500 (O)	0	0.96580	0.03000	0.96790	0.01340

Table 8.15: (Triple) Injection points efficiencies for rule induction with sampling

Data Set	Sampling	N	TPR	FPR	AUC	SD
crc	500 (O)	2	0.96918	0.01931	0.97542	0.01782
dijkstra	100 (O)	6	0.98967	0.00888	0.99089	0.00911
fft	300 (O)	6	0.93077	0.05791	0.93687	0.00160
search	200 (O)	8	0.95516	0.03706	0.95951	0.04421
insert	500 (O)	3	0.96605	0.00473	0.98115	0.01166
remove	400 (O)	3	0.98558	0.01027	0.98814	0.02930
encfile	600 (O)	3	0.97070	0.00030	0.98570	0.00416
decfile	600 (O)	2	0.99060	0.05454	0.96847	0.00160
isqrt	700 (U)	9	0.93057	0.04578	0.94285	0.01029
cubic	600 (O)	0	0.96677	0.02997	0.96887	0.01341

Despite being the best performing algorithm under a baseline configuration, the entries in Tables 8.17-8.19 show consistent improvements, with respect to the

Table 8.16: (Quadruple) Injection points efficiencies for rule induction with sampling

Data Set	Sampling	N	TPR	FPR	AUC	SD
crc	500 (O)	2	0.97015	0.01929	0.97640	0.01784
dijkstra	100 (O)	6	0.99066	0.00887	0.99188	0.00912
fft	300 (O)	6	0.93170	0.05785	0.93781	0.00161
search	200 (O)	8	0.95611	0.03702	0.96047	0.04425
insert	500 (O)	3	0.96701	0.00472	0.98213	0.01167
remove	400 (O)	3	0.98657	0.01026	0.98913	0.02933
encfile	600 (O)	3	0.97167	0.00030	0.98669	0.00417
decfile	600 (O)	2	0.99159	0.05449	0.96944	0.00160
isqrt	700 (O)	9	0.93150	0.04574	0.94379	0.01030
cubic	600 (O)	0	0.96773	0.02994	0.96984	0.01343

mean AUC measure, during the fault injection point predictive models optimisation process. Though, in some cases this improvement is small. In almost all cases the standard deviation of all models is marginally decreased.

Table 8.17: (Double) Injection points efficiencies for decision tree induction with sampling

Data Set	Sampling	N	TPR	FPR	AUC	SD
crc	600 (O)	8	0.99538	0.00047	0.99746	0.00290
dijkstra	800 (U)	4	0.99635	0.00007	0.99815	0.00016
fft	300 (O)	9	0.99860	0.00002	0.99929	0.00013
search	300 (U)	0	0.86068	0.00452	0.92988	0.00150
insert	200 (U)	6	0.99684	0.00004	0.99840	0.00067
remove	700 (U)	7	0.99578	0.00176	0.99702	0.00046
encfile	100 (O)	2	0.99968	0.00665	0.99652	0.00004
decfile	15 (O)	0	0.97362	0.00000	0.98682	0.00000
isqrt	50 (U)	2	0.99551	0.00000	0.99775	0.00000
cubic	400 (O)	0	0.97940	0.00160	0.98890	0.00060

8.6.4 Bit-position and injection efficiency

As mentioned, the fault injection datasets are generated from fault injection analysis that sampled random bit-positions, i.e., faults are injected randomly. As such the injection efficiency of the generated fault injection points, to this point, has been based on this sample. This will serve as a baseline to assess the validity of the approach of discerning efficient variable-bit combinations for fault

Table 8.18: (Triple) Injection points efficiencies for decision tree induction with sampling

Data Set	Sampling	N	TPR	FPR	AUC	SD
crc	500 (O)	9	0.99558	0.00047	0.99766	0.00287
dijkstra	100 (O)	4	0.99655	0.00007	0.99835	0.00016
fft	300 (O)	9	0.99880	0.00002	0.99949	0.00013
search	200 (U)	0	0.86086	0.00452	0.93007	0.00145
insert	500 (U)	6	0.99704	0.00004	0.99860	0.00065
remove	400 (U)	7	0.99598	0.00176	0.99722	0.00043
encfile	600 (O)	2	0.99988	0.00665	0.99672	0.00003
decfile	600 (O)	0	0.97382	0.00000	0.98701	0.00000
isqrt	700 (O)	2	0.99571	0.00000	0.99795	0.00000
cubic	600 (O)	0	0.97960	0.00160	0.98910	0.00059

Table 8.19: (Quadruple) Injection points efficiencies for decision tree induction with sampling

Data Set	Sampling	N	TPR	FPR	AUC	SD
crc	500 (O)	9	0.99568	0.00047	0.99776	0.00283
dijkstra	100 (O)	4	0.99665	0.00007	0.99845	0.00011
fft	300 (O)	9	0.99890	0.00002	0.99959	0.00010
search	200 (O)	0	0.86094	0.00452	0.93016	0.00141
insert	500 (O)	6	0.99714	0.00004	0.99870	0.00030
remove	400 (O)	7	0.99608	0.00176	0.99732	0.00041
encfile	600 (O)	2	0.99998	0.00664	0.99682	0.00002
decfile	600 (O)	0	0.97392	0.00000	0.98711	0.00000
isqrt	700 (O)	2	0.99581	0.00000	0.99805	0.00000
cubic	600 (O)	0	0.97969	0.00160	0.98920	0.00055

injection. As the computation cost of performing an exhaustive bit-flipping for more than two faults is prohibitive, the comparisons is done using single and double faults. Thus using the the process explained in Section 8.6.1, a second version of double-bits dataset is generated from exhaustive bit-flipping for each case study. Following the generation of the datasets, they are processed as explained in Sections 8.6.2-8.6.3. The results of the performance of the predictive models (baseline models) generated by the process in Section 8.6.2 are presented in Tables 8.20-8.22, and those (optimised models) by the process in Section 8.6.3 are shown in Tables 8.23-8.25. In this section, these new datasets, will be referred to as *exhaustive (double-bits) datasets* and the double-bits used in previous section will be called *random (double-bits) datasets.* Before pro-

ceeding with the analysis of the results, the key finding of these results can be summarised as follows:

- fault injection points predictive models generated from a subset of the bit-positions can be as efficient as those generated from the entire bit-positions set.

It has been observed from Tables 8.2-8.17 and Tables 8.20-8.25 that the difference in the injection efficiency of the fault injection points predictive models generated using exhaustive bit-flipping and those generated using random bit-flipping is small. The largest difference in AUC when comparing these results is associated with datasets *fft* (baseline and optimised naïve Bayes models), *insert* (baseline rule induction), *dijstra* (optimised rule induction), *decfile* (baseline decision tree induction) and *isqrt* (optimised decision tree induction). For these dataset the predictive models generated using exhaustive bit-flipping have a mean AUC of 0.95884, 0.96619, 0.98337, 0.99188, 0.99866 and 0.99975, whilst those generated using random bit-flipping have a mean AUC of 0.9474, 0.96429, 0.97171, 0.98990, 0.98681 and 0.99775 giving a difference of just 0.01103, 0.00193, 0.01166, 0.00198, 0.01184 and 0.00200 in these worst cases, for baseline naïve Bayes, optimise naïve Bayes, baseline rule induction, optimise rule induction, baseline decision tree induction and optimised decision tree induction respectively.

It should be observed also that the absolute AUC values for fault injection point predictive models generated using only a subset of bit are consistently high, with the maximum and minimum for baseline naïve Bayes being 0.94747 and 0.89276 respectively, for optimised naïve Bayes being 0.96426 and 0.92365 respectively, for baseline rule induction being 0.97171 and 0.92252 respectively, for optimised rule induction being 0.98990 and 0.93594 respectively, for baseline decision tree induction being 0.99908 and 0.92888 respectively and for optimised decision tree induction being 0.99929 and 0.92988 respectively. These consistently high

Table 8.20: (Double) Injection points efficiencies for naïve Bayes with no sampling using full bit set

Data Set	TPR	FPR	AUC	SD
crc	0.87056	0.03082	0.92549	0.02706
dijkstra	0.95016	0.09877	0.93050	0.01826
fft	0.91585	0.00993	0.95884	0.03453
search	0.96249	0.11730	0.92717	0.01418
insert	0.87788	0.04927	0.91971	0.01179
remove	0.87976	0.06963	0.91022	0.03854
encfile	0.87389	0.05329	0.91565	0.03605
decfile	0.91429	0.11652	0.90347	0.03804
isqrt	0.90791	0.04379	0.93753	0.03139
cubic	0.90584	0.06550	0.92537	0.02776

Table 8.21: (Double) Injection points efficiencies for rule induction with no sampling using full bit set

Data Set	TPR	FPR	AUC	SD
crc	0.95966	0.01909	0.97605	0.00011
dijkstra	0.94119	0.03110	0.96067	0.00214
fft	0.93425	0.07719	0.93359	0.00025
search	0.96228	0.04991	0.96158	0.00039
insert	0.98991	0.03434	0.98337	0.00114
remove	0.97642	0.03882	0.97770	0.00044
encfile	0.95054	0.01695	0.97259	0.00558
decfile	0.99136	0.05503	0.97350	0.00016
isqrt	0.94080	0.04528	0.95321	0.00016
cubic	0.96069	0.04090	0.96585	0.00119

Table 8.22: (Double) Injection points efficiencies for decision tree induction with no sampling using full bit set

Data Set	TPR	FPR	AUC	SD
crc	0.97085	0.00004	0.99141	0.00572
dijkstra	0.99902	0.00142	0.99583	0.00017
fft	0.99967	0.00001	0.99998	0.00012
search	0.86815	0.00009	0.94003	0.00061
insert	0.99833	0.00010	0.99508	0.00009
remove	0.99822	0.00131	0.99441	0.00123
encfile	0.99175	0.00103	0.99046	0.00011
decfile	0.98531	0.00000	0.99866	0.00000
isqrt	0.99700	0.00011	0.99870	0.00014
cubic	0.98134	0.00049	0.99642	0.00089

AUC values, which are indicative of high true positive and low false positive rates, serve to suggest that fault injection points generated using sample bits can safeguard the functioning of a software system. Further, the fact that standard deviation in AUC remains low, even unchanged in many cases, when only random bits are used in the generation of fault injection points means that efficient injection points can be consistently generated across separate cross validations.

Table 8.23: (Double) Injection points efficiencies for naïve Bayes with sampling using full bit set

Data Set	**Sampling**	**N**	**TPR**	**FPR**	**AUC**	**SD**
crc	500 (O)	8	0.91013	0.00362	0.95425	0.05581
dijkstra	1000 (O)	4	0.95427	0.08112	0.93742	0.03021
fft	300 (O)	3	0.93250	0.00211	0.96619	0.17172
search	200 (O)	2	0.97387	0.05776	0.95894	0.05458
insert	500 (O)	5	0.90728	0.05805	0.92550	0.05158
remove	400 (O)	8	0.91889	0.01362	0.95361	0.04182
encfile	600 (O)	8	0.89617	0.02537	0.93635	0.05295
decfile	600 (O)	6	0.94927	0.05716	0.94694	0.06359
isqrt	600 (O)	7	0.92295	0.00475	0.96009	0.08223
cubic	500 (O)	9	0.92565	0.03134	0.94809	0.06756

Table 8.24: (Double) Injection points efficiencies for rule induction with sampling using full bit set

Data Set	**Sampling**	**N**	**TPR**	**FPR**	**AUC**	**SD**
crc	700 (O)	2	0.97015	0.01929	0.97640	0.01776
dijkstra	500 (O)	6	0.99066	0.00887	0.99188	0.00908
fft	300 (O)	6	0.93170	0.05785	0.93781	0.00160
search	400 (O)	8	0.95611	0.03702	0.96047	0.04408
insert	700 (O)	3	0.96701	0.00472	0.98213	0.01162
remove	600 (O)	3	0.98657	0.01026	0.98913	0.02921
encfile	500 (O)	3	0.97167	0.00030	0.98669	0.00415
decfile	400 (O)	2	0.99159	0.05449	0.96944	0.00160
isqrt	400 (O)	9	0.93150	0.04574	0.94379	0.01026
cubic	500 (O)	10	0.96773	0.02994	0.96984	0.01337

This implies that the proposed approach remains robust when using random bits, which is particularly important given that using datasets containing fewer bits, in effect, reduces the amount of information available during the construction of fault injection points for multiple soft-errors. This substantiate the thesis

Table 8.25: (Double) Injection points efficiencies for decision tree induction with sampling using full bit set

Data Set	Sampling	N	TPR	FPR	AUC	SD
crc	600 (O)	8	0.99737	0.00047	0.99945	0.00283
dijkstra	800 (O)	4	0.99835	0.00007	0.99914	0.00010
fft	300 (O)	9	0.99990	0.00002	0.99999	0.00009
search	300 (O)	5	0.86240	0.00451	0.93174	0.00139
insert	200 (O)	6	0.99702	0.00004	0.99858	0.00029
remove	700 (O)	7	0.99777	0.00175	0.99901	0.00039
encfile	100 (O)	2	0.99998	0.00663	0.99851	0.00002
decfile	15 (U)	-	0.97557	0.00000	0.98879	0.00000
isqrt	50 (U)	-	0.99750	0.00000	0.99975	0.00000
cubic	400 (O)	2	0.98136	0.00160	0.99088	0.00053

claim that perturbing certain bits in combination of variables is as efficient as performing an exhaustive perturbation in all variable-bit-wise combinations.

8.7 Implication and Limitation

The case studies presented have demonstrated the applicability of the proposed approach is terms of generating predictive models for selecting efficient fault injection points for multiple soft-errors. In particular, decision tree induction and rule induction have, even under a baseline configuration, been shown to be effective and consistent methods for generating predictive models for detecting failure-inducing points which exhibit high coverage. In the case of decision tree induction and rule induction, generated predictive models are represented as a tree structure to be interpreted as a conjunction of disjunctions and as a first-order predicate respectively.

Despite the presented case studies suggesting that the decision tree induction and rule induction algorithms yield significantly more efficient fault injection points predictive models than naïve Bayes, it is not possible to conclude that these algorithms will consistently outperform other algorithms, including naïve Bayes. As any two classification algorithms can differ only in the class boundary

that they define, i.e., the boundary defined to classify system failure classes and non-failures in the generation of fault injection points predictive models, it is not possible to determine which classification algorithm will define an boundary that is appropriate for a particular dataset. Indeed, it is current practise in data mining approaches to classification problems to seek out an acceptable model through the investigation of many classification algorithms.

Even though this a approach is proposed to improve and complement and the framework propose in Chapte 7, it is not tied to the framework, this means this approach can be applied to any fault injection datasets regardless of the method used in selecting the target program locations. This implies that the main cost of applying the proposed approach is associated with the execution of data mining algorithms, which in-turn implies that the cost of generating efficient fault injection points using the approach is related to dataset magnitude, the data mining algorithm applied and the comprehensiveness of the optimisation undertaken, i.e., the number of algorithm configurations that are considered in model optimisation. It was shown in the cases studies presented in Section 8.6.2 that using only a baseline configuration of several data mining algorithms can yield highly-efficient failure-inducing injection points and that systematically varying the level of sampling applied to datasets, can allow the efficiency of those injection points to be consistently improved, often to levels that would make them applicable in the validation of dependable software systems. Further, as with any approach that uses fault injection data, the efficacy of the proposed model is constrained by the assumed fault model and input set used in the fault inject analysis the data is derived from.

8.8 Summary and Conclusions

In this chapter an approach to systematically reduce fault space for multiple soft-errors injections has been proposed. The intuition of the proposed approach is

that, given a set of target program locations at which multiple bit-flip faults will be injected, data mining methods can be applied to fault injection datasets to identify efficient set of bit-positions that will induce as much system failures as if the entire bit-position fault space has been used. Following its descriptions, the proposed approach was applied to ten embedded software modules, for each of these modules, fault injection points for multiple target locations was generated and their efficiency evaluated. The results demonstrated that the proposed approach can be effectively used to identify a number of bits to flip from a set of target program variables, that will cause almost the same amount of system failure if the entire bits space is considered, i.e., the injection points exhibit high coverage.

CHAPTER 9

Conclusions

To this point, this thesis presents research, analysis and discussions to substantiate the thesis statement:

> **"There exists a computational feasible bits to explore under multiple bit-flip faults that will induce a wider failure profile."**

In this chapter a summary of these research contributions and a discussion of future work relating to the exponential multiple fault space problem is provided as a conclusion to this thesis. In particular, the research contributions made throughout Chapters 5-8 are summarised with respect to the stated thesis.

9.1 Research Contribution Summary

In support of the stated thesis, the following specific contributions were made to the selection of efficient fault injection points for multiple soft-errors, more detailed account can be found in the respective chapters as indicated.

9.1.1 Complexity Analysis and Formalisation of ILS and TVS Problem

To circumvent around the exponential size of the fault space for multiple soft-errors, Chapter 5 analysis the the complexity of the following sub-problems: (i) choosing the locations in which faults can be injected and (ii) choosing the variables in which faults will be injected. Each problem is formally defined as an NP-problem and two known NP-complete problems, MVC and MDC, respectively, were reduced to the former and latter sub-problems to prove their NP-completeness respectively.

9.1.2 Double Single Bit-Flips Fault Model

In Chapter 6, a novel fault model representative of emerging hardware faults due to technology advances is proposed. The chapter investigated the impact of such fault once translated into soft errors. The model extend the traditional model of simple faults to double faults in combination of two locations. The usability of the proposed model for software dependability validation is evaluated using fault injection analysis. The results show that the proposed model induces a different failure profile compared with that caused by single fault model and an existing variant double fault model.

9.1.3 Heuristics for the Injection Locations and Target Variables Selection

Chapter 7 presents a framework for selection of efficient fault injection locations (in terms of potential injection location and target variable) in a program. The approach takes account of relationships between block locations, program variables and program states. The selection is done by applying static analysis

and graph theory concepts on the software byte code, to first, discern potential injection location and then to determine the most suitable combination of variables to target within these locations. The framework provided a systematic approach for the selection component of the L_nC_m fault model. Furthermore, the research evaluated the approach and validity of the L_nC_m fault model on several case studies.

9.1.4 Efficient Bit Locations

In Chapter 8, a systematic approach for the selection of efficient fault injection points for multiple soft-errors is proposed, based on the application of data mining approaches to datasets generated from fault injection analysis. This is done in order to refine the exponential size of the fault injection points space (in terms of variables and bits combinations). The results demonstrate that a subset of bits within a set of given locations can achieve similar injection efficiency to that of the entire bits set. This results serve to substantiate the thesis which this research is based on.

9.2 Applications

The work presented in this thesis can be used by system developers and engineers for the development and validation of software-implemented hardware fault tolerance techniques (SIHFT). For instance it can be used to aid the design and evaluation of detectors for multiple soft-errors. The research can be used by individuals and organisations for experimental benchmarking of the error sensitivity of software components. Such benchmarking experiments is done in order to measure the likelihood that the executable code of a software component exhibit silent data corruptions (SDCs) for hardware errors that propagate to ISA registers and main memory locations. The purpose of such measurements

is to identify weaknesses in the executable code, and thereby finding ways of hardening the code against hardware errors by means of SIHFT.

9.3 Future Work

The selection of efficient fault injection locations for multiple soft-errors remains a key challenge in the development of fault tolerant software systems, particularly in the content of real-world, embedded software systems. In fact, there is scarcity of research investigating systematic approaches in selecting injection points for multiple fault injections. Despite the progress made by the work presented in this thesis, there are many areas for future work relating to the problem of multiple soft-errors injections. Fews areas for future research relating to the work presented in this thesis are summarised as follows:

- Typical to embedded systems, programs goes though different iterations as they execute. This produces a dominator graph instead of a dominator tree. Capturing this structure is important, if the notion of amplification is to effectively encapsulated for such systems. As mentioned in previous chapters, this thesis, have not specifically address the problem of handling looping structures in program. Since the approach for selecting potential locations is based on a dominator tree, then there is no cycle in such a structure, meaning that the heuristic will terminate with a set of potential injection locations. Given that the heuristic for selecting variables is also obtained from the dominator tree and that there are no loops in the resulting dependency graph, then the heuristic will terminate properly as well. Thus future work will look into the design amplification metric to decide when to inject and when not to inject faults through the different iterations of a programs execution especially for embedded control systems in order to enhance the applicability of the framework presented in this

thesis. To Further enhance the effectiveness of the approach presented in Chapters 5-8, comparative studies with different compiler optimisations, hardware platforms, and different programming languages may be considered. Another important part of this work is to extend the study with experiments on target programs that are equipped with SIHFT techniques will be conducted.

- There are software that cannot be modelled as a control flow graph, such as operating systems and device drivers [163]. Since applications running on such systems make system calls to execute, we believe that such calls can be abstracted in an inter-procedural CFG. However, the notion of dominance and dominating graph to select best parameters (in a system call) as target variables may not be suited to such environments. This is an area considered for future research.
- The work indicates there are intuitions that can guide the selection of bit-positions. As future work, such intuitions for bits-selection, will be investigates, analysed, documented and developed.

Bibliography

[1] F. Adamu-Fika and A. Jhumka. *Algorithms and Architectures for Parallel Processing: 15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18-20, 2015, Proceedings, Part IV*, chapter An Investigation of the Impact of Double Bit-Flip Error Variants on Program Execution, pages 799–813. Springer International Publishing, Cham, 2015. ISBN 978-3-319-27140-8. doi: 10.1007/978-3-319-27140-8_55. URL http://dx.doi.org/10.1007/978-3-319-27140-8_55.

[2] F. Adamu-Fika and A. Jhumka. An investigation of the impact of double single bit-flip errors on program executions. In P. Lorenz and F. P. Dini, editors, *DEPEND 2015, The Eight International Conference on Dependability*, pages 15 – 22, Venice, Italy, August 2015. IARIA. ISBN 978-1-61208-429-9. URL http://www.thinkmind.org/index.php?view=article&articleid=depend_2015_1_40_50038.

[3] F. Adamu-Fika and A. Jhumka. Towards learning bit patterns, 2015. [Under submission to IEEE Transactions on Dependable and Secure Computing, Special Issue on Data-Driven Dependability and Security].

[4] F. Adamu-Fika and A. Jhumka. Towards the efficient selection of injection locations for multiple transient hardware faults, 2015. [Unpublished].

[5] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. Softw. Eng.*, 16(2): 166–182, Feb. 1990. ISSN 0098-5589. doi: 10.1109/32.44380. URL http://dx.doi.org/10.1109/32.44380.

[6] A. Arora and S. S. Kulkarni. Detectors and correctors: a theory of fault-tolerance components. In *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*, pages 436–443, May 1998. doi: 10.1109/ICDCS.1998.679772.

[7] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan 2004. ISSN 1545-5971. doi: 10.1109/TDSC.2004.2.

[8] S. Ayache, E. Conquet, P. Humbert, C. Rodriguez, J. Sifakis, and R. Gerlich. Formal methods for the validation of fault tolerance in autonomous spacecraft. In *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, pages 353–357, Jun 1996. doi: 10.1109/FTCS.1996.534620.

[9] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson. A study of the impact of single bit-flip and double bit-flip errors on program execution. In F. Bitsch, J. Guiochet, and M. Kaniche, editors, *Computer Safety, Reliability, and Security*, volume 8153 of *Lecture Notes in Computer Science*, pages 265–276. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40792-5. doi: 10.1007/978-3-642-40793-2_24. URL `http://dx.doi.org/10.1007/978-3-642-40793-2_24`.

[10] A. Bordes, S. Ertekin, J. Weston, and L. Bottou. Fast kernel classifiers with online and active learning. *J. Mach. Learn. Res.*, 6:1579–1619, Dec. 2005. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=1046920.1194898`.

[11] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6): 10–16, Nov. 2005. ISSN 0272-1732. doi: 10.1109/MM.2005.110. URL `http://dx.doi.org/10.1109/MM.2005.110`.

[12] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Statistics/Probability Series. Wadsworth Publishing Company, Belmont, California, U.S.A., 1984.

[13] C. Brunk and M. J. Pazzani. Proceedings of the an investigation of noise-tolerant relational concept learning algorithms. In *8th International Conference on Machine Learning*, 1991.

[14] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for internet applications. In *Proceedings of the The Third IEEE Workshop on Internet Applications*, WIAPP '03, pages 132–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1972-5. URL `http://dl.acm.org/citation.cfm?id=832311.837386`.

[15] E. H. Cannon, M. S. Gordon, D. F. Heidel, A. KleinOsowski, P. Oldiges, K. P. Rodbell, and H. H. Tang. Multi-bit upsets in 65nm SOI SRAMs. In *Reliability Physics Symposium, 2008. IRPS 2008. IEEE International*, pages 195–201, April 2008. doi: 10.1109/RELPHY.2008.4558885.

[16] J. Carreira, H. Madeira, and J. G. Silva. Xception: Software fault injection and monitoring in processor functional units. In *Proceedings of the Fifth IFIP Working Conference on Dependable Computing for Critical Applications*, pages 135–149, September 1998.

[17] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, June 2002. ISSN 1076-9757. URL `http://dl.acm.org/citation.cfm?id=1622407.1622416`.

[18] N. V. Chawla, D. A. Cieslak, L. O. Hall, and A. Joshi. Automatically countering imbalance and its empirical relationship to cost. *Data Mining and Knowledge Discovery*, 17(2):225–252, 2008. ISSN 1573-756X.

doi: 10.1007/s10618-008-0087-0. URL http://dx.doi.org/10.1007/s10618-008-0087-0.

[19] H. Cho, S. Mirkhani, C. Y. Cher, J. A. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–10, May 2013.

[20] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante. Exploiting fpga for accelerating fault injection experiments. In *On-Line Testing Workshop, 2001. Proceedings. Seventh International*, pages 9–13, 2001. doi: 10.1109/OLT.2001.937810.

[21] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante. Exploiting fpga for accelerating fault injection experiments. In *On-Line Testing Workshop, 2001. Proceedings. Seventh International*, pages 9–13, 2001. doi: 10.1109/OLT.2001.937810.

[22] W. W. Cohen. Fast effective rule induction. In *Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

[23] J. Cong and K. Gururaj. Assuring application-level correctness against soft errors. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, pages 150–157, Piscataway, NJ, USA, 2011. IEEE Press. ISBN 978-1-4577-1398-9. URL http://dl.acm.org/citation.cfm?id=2132325.2132360.

[24] J. Cong and K. Gururaj. Assuring application-level correctness against soft errors. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pages 150–157, November 2011. doi: 10.1109/ICCAD.2011.6105319.

[25] S. A. Cook. An overview of computational complexity. *Commun. ACM*, 26(6):400–408, June 1983. ISSN 0001-0782. doi: 10.1145/358141.358144. URL http://doi.acm.org/10.1145/358141.358144.

[26] D. Cotroneo and R. Natella. Fault injection for software certification. *Security Privacy, IEEE*, 11(4):38–45, July 2013. ISSN 1540-7993. doi: 10.1109/MSP.2013.54.

[27] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512973. URL `http://doi.acm.org/10.1145/512950.512973`.

[28] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 497–508, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. doi: 10.1145/1815961.1816026. URL `http://doi.acm.org/10.1145/1815961.1816026`.

[29] T. A. Delong, B. W. Johnson, and J. A. Profeta. A fault injection technique for vhdl behavioral-level models. *IEEE Design Test of Computers*, 13(4):24–33, Winter 1996. ISSN 0740-7475. doi: 10.1109/54.544533.

[30] M. Demertzi, M. Annavaram, and M. Hall. Analyzing the effects of compiler optimizations on application reliability. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 184–193, November 2011. doi: 10.1109/IISWC.2011.6114178.

[31] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 233–244, New York, NY, USA, 2006. ACM. ISBN 1-59593-263-1. doi: 10.1145/1146238.1146266. URL `http://doi.acm.org/10.1145/1146238.1146266`.

[32] D. Di Leo, F. Ayatolahi, B. Sangchoolie, J. Karlsson, and R. Johansson. On the impact of hardware faults — an investigation of the relationship between workload inputs and failure mode distributions. In *Proceedings of the 31st International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'12, pages 198–209, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-33677-5. doi: 10.1007/978-3-642-33678-2_17. URL http://dx.doi.org/10.1007/978-3-642-33678-2_17.

[33] A. Dixit and A. Wood. The impact of new technology on soft error rates. In *Reliability Physics Symposium (IRPS), 2011 IEEE International*, pages 5B.4.1–5B.4.7, April 2011. doi: 10.1109/IRPS.2011.5784522.

[34] P. Domingos. Metacost: A general method for making classifiers cost-sensitive. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, pages 155–164, New York, NY, USA, 1999. ACM. ISBN 1-58113-143-7. doi: 10.1145/312129.312220. URL http://doi.acm.org/10.1145/312129.312220.

[35] C. Elkan. The foundations of cost-sensitive learning. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'01, pages 973–978, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-812-5, 978-1-558-60812-2. URL http://dl.acm.org/citation.cfm?id=1642194.1642224.

[36] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0. doi: 10.1145/302405.302467. URL http://doi.acm.org/10.1145/302405.302467.

[37] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S.

Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming, Special Issue on Experimental Software and Toolkits*, 69(3):35–45, December 2007.

[38] S. Ertekin, J. Huang, L. Bottou, and L. Giles. Learning on the border: Active learning in imbalanced data classification. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, pages 127–136, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-803-9. doi: 10.1145/1321440.1321461. URL `http://doi.acm.org/10.1145/1321440.1321461`.

[39] S. Ertekin, J. Huang, and C. L. Giles. Active learning for class imbalance problem. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, pages 823–824, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-597-7. doi: 10.1145/1277741.1277927. URL `http://doi.acm.org/10.1145/1277741.1277927`.

[40] J.-C. Fabre, M. Rodriguez, J. Arlat, and J.-M. Sizun. Building dependable cots microkernel-based systems using mafalda. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependalbe Computing*, pages 85–92, August 2000.

[41] W. Fan, S. J. Stolfo, J. Zhang, and P. K. Chan. Adacost: Misclassification cost-sensitive boosting. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 97–105, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-612-2. URL `http://dl.acm.org/citation.cfm?id=645528.657651`.

[42] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 385–396, New York, NY, USA,

2010. ACM. ISBN 978-1-60558-839-1. doi: 10.1145/1736020.1736063. URL `http://doi.acm.org/10.1145/1736020.1736063`.

[43] C. Fetzer and Z. Xiao. An automated approach to increasing the robustness of c libraries. In *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 155–164, December 2002.

[44] G. Fey, A. Sulflow, and R. Drechsler. Computing bounds for fault tolerance using formal techniques. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 190–195, July 2009. doi: 10.1145/1629911.1629963.

[45] P. Folkesson, S. Svensson, and J. Karlsson. A comparison of simulation based and scan chain implemented fault injection. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 284–293, June 1998. doi: 10.1109/FTCS.1998.689479.

[46] E. Fuchs. An evaluation of the error detection mechanisms in mars using software-implemented fault injection. In *Proceedings of the Second European Dependable Computing Conference on Dependable Computing*, EDCC-2, pages 73–90, London, UK, UK, 1996. Springer-Verlag. ISBN 3-540-61772-8. URL `http://dl.acm.org/citation.cfm?id=645331.649805`.

[47] J. Fürnkranz and G. Widmer. Incremental Reduced Error Pruning. In W. W. Cohen and H. Hirsh, editors, *Proceedings of the 11th International Conference on Machine Learning (ML-94)*, pages 70–77, New Brunswick, NJ, 1994. Morgan Kaufmann. URL `http://www.ke.informatik.tu-darmstadt.de/~juffi/publications/ml-94.ps.gz`.

[48] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 2000.

[49] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

[50] F. C. Gärtner and A. Jhumka. Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. In Y. Lakhnech and S. Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 183–198. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23167-7. doi: 10.1007/978-3-540-30206-3_14. URL `http://dx.doi.org/10.1007/978-3-540-30206-3_14`.

[51] G. Georgakos, P. Huber, M. Ostermayr, E. Amirante, and F. Ruckerbauer. Investigation of increased multi-bit failure rate due to neutron induced SEU in advanced embedded srams. In *VLSI Circuits, 2007 IEEE Symposium on*, pages 80–81, June 2007. doi: 10.1109/VLSIC.2007.4342774.

[52] J. Grinschgl, A. Krieg, C. Steger, R. Weiss, H. Bock, J. Haid, T. Aichinger, and C. Ulbricht. Case study on multiple fault dependability and security evaluations. *Microprocessors and Microsystems*, 37(2):218 – 227, 2013. ISSN 0141-9331. doi: http://dx.doi.org/10.1016/j.micpro.2012.05.016. URL `http://www.sciencedirect.com/science/article/pii/S0141933112000932`. Digital System Safety and Security.

[53] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7803-7315-4. doi: 10.1109/WWC.2001.15. URL `http://dx.doi.org/10.1109/WWC.2001.15`.

[54] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H.

Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009. ISSN 1931-0145. doi: 10.1145/1656274.1656278. URL http://doi.acm.org/10.1145/1656274.1656278.

[55] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-1624-8. URL http://dl.acm.org/citation.cfm?id=2354410.2355132.

[56] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Trans. on Knowl. and Data Eng.*, 21(9):1263–1284, Sept. 2009. ISSN 1041-4347. doi: 10.1109/TKDE.2008.239. URL http://dx.doi.org/10.1109/TKDE.2008.239.

[57] M. Hiller. Executable assertions for detecting data errors in embedded control systems. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 24–33, 2000. doi: 10.1109/ICDSN.2000.857510.

[58] M. Hiller, A. Jhumka, and N. Suri. An approach for analysing the propagation of data errors in software. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 161–170, July 2001. doi: 10.1109/DSN.2001.941402.

[59] M. Hiller, A. Jhumka, and N. Suri. PROPANE: An environment for examining the propagation of errors in software. In *Proceedings of the 11th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 81–85, July 2002.

[60] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. In *Dependable Systems and Networks,*

2002. DSN 2002. Proceedings. International Conference on, pages 135–144, 2002. doi: 10.1109/DSN.2002.1028894.

[61] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, Apr 1997. ISSN 0018-9162. doi: 10.1109/2.585157.

[62] ISO. ISO 26262-1:2011, road vehicles – functional safety – part 1: Vocabulary, 2011.

[63] N. Japkowicz. The class imbalance problem: Significance and strategies. In *Proceedings of the 2000 International Conference on Artificial Intelligence (ICAI*, pages 111–117, 2000.

[64] A. Jhumka and M. Hiller. Putting detectors in their place [program monitoring]. In *Software Engineering and Formal Methods, 2005. SEFM 2005. Third IEEE International Conference on*, pages 33–42, September 2005. doi: 10.1109/SEFM.2005.38.

[65] A. Jhumka and M. Leeke. Issues on the design of efficient fail-safe fault tolerance. In *Software Reliability Engineering, 2009. ISSRE '09. 20th International Symposium on*, pages 155–164, Nov 2009. doi: 10.1109/ISSRE.2009.31.

[66] A. Jhumka and M. Leeke. The early identification of detector locations in dependable software. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 40–49, Nov 2011. doi: 10.1109/ISSRE.2011.34.

[67] A. Jhumka, M. Hiller, and N. Suri. Assessing inter-modular error propagation in distributed software. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, pages 152–161, January 2001.

[68] A. Jhumka, F. C. Gärtner, C. Fetzer, and N. Suri. On systematic design of fast and perfect detectors. Technical Report 200263, Swiss Federal

Institute of Technology (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland, Sept. 2002.

[69] A. Jhumka, M. Hiller, and N. Suri. An approach for designing and assessing detectors for dependable component-based systems. In *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*, pages 69–78, March 2004. doi: 10.1109/HASE.2004.1281731.

[70] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, 2009.

[71] G. H. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, UAI'95, pages 338–345, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-385-9. URL `http://dl.acm.org/citation.cfm?id=2074158.2074196`.

[72] P. Joshi, H. Gunawi, and K. Sen. Prefail: a programmable tool for multiple-failure injection. In *Proceedings OOPSLA*, 2011.

[73] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2):248–260, February 1995.

[74] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro*, 14(1):8–11, 13–23, Feb. 1994. ISSN 0272-1732. doi: 10.1109/40.259894. URL `http://dx.doi.org/10.1109/40.259894`.

[75] R. M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972. ISBN 978-1-4684-2001-2. doi: 10.1007/978-1-4684-2001-2_9. URL `http://dx.doi.org/10.1007/978-1-4684-2001-2_9`.

[76] D. S. Khudia, G. Wright, and S. Mahlke. Efficient soft error protection for commodity embedded microprocessors using profile information. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 99–108, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1212-7. doi: 10.1145/2248418.2248433. URL `http://doi.acm.org/10.1145/2248418.2248433`.

[77] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. C. Hoe. Multi-bit error tolerant caches using two-dimensional error coding. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 197–209, December 2007. doi: 10.1109/MICRO.2007.19.

[78] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *SIGARCH Comput. Archit. News*, 42(3):361–372, June 2014. ISSN 0163-5964. doi: 10.1145/2678373.2665726. URL `http://doi.acm.org/10.1145/2678373.2665726`.

[79] P. Koopman, K. DeVale, and J. DeVale. Interface robustness testing: experiences and lessons learned from the ballista project. In K. Kanoun and L. Spainhower, editors, *Dependability Benchmarking for Computer Systems*, pages 201–226. IEEE Press, 2008.

[80] M. Kubat and S. Matwin. Addressing the curse of imbalanced training sets: One-sided selection. In *In Proceedings of the Fourteenth International Conference on Machine Learning*, pages 179–186. Morgan Kaufmann, 1997.

[81] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Computer Science*, pages 82–93.

Springer Berlin Heidelberg, 2000. ISBN 978-3-540-41055-3. doi: 10.1007/3-540-45352-0_9. URL http://dx.doi.org/10.1007/3-540-45352-0_9.

[82] S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 337–344, 2002. doi: 10.1109/ICDCS.2002.1022271.

[83] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri. An empirical study of injected versus actual interface errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 397–408, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2645-2. doi: 10.1145/2610384.2610418. URL http://doi.acm.org/10.1145/2610384.2610418.

[84] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[85] S. le Cessie and J. van Houwelingen. Ridge estimators in logistic regression. *Applied Statistics*, 41(1):191–201, 1992.

[86] M. B. Lecocke, J. Blount, and J. Blount. Use of formal modeling to automatically generate correct fault detection and response methods. In *2015 IEEE Aerospace Conference*, pages 1–7, March 2015. doi: 10.1109/AERO.2015.7119245.

[87] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Partially protected caches to reduce failures due to soft errors in multimedia applications. *Very Large Scale Integration (VLSI) Systems,*

IEEE Transactions on, 17(9):1343–1347, Sept 2009. ISSN 1063-8210. doi: 10.1109/TVLSI.2008.2002427.

[88] W. Lee, S. J. Stolfo, and K. W. Mok. A data mining framework for building intrusion detection models. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 120–132, 1999. doi: 10.1109/SECPRI.1999.766909.

[89] M. Leeke and A. Jhumka. Towards understanding the importance of variables in dependable software. In *Dependable Computing Conference (EDCC), 2010 European*, pages 85–94, April 2010. doi: 10.1109/EDCC.2010.20.

[90] M. Leeke, S. Arif, A. Jhumka, and S. S. Anand. A methodology for the generation of efficient error detection mechanisms. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 25–36, June 2011. doi: 10.1109/DSN.2011.5958204.

[91] M. Leeke, A. Jhumka, and S. S. Anand. Towards the design of efficient error detection mechanisms for transient data errors. *The Computer Journal*, 56(6):674–692, 2013.

[92] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[93] N. Leveson, S. Cha, J. Knight, and T. Shimeall. The use of self checks and voting in software error detection: an empirical study. *Software Engineering, IEEE Transactions on*, 16(4):432–443, April 1990. ISSN 0098-5589. doi: 10.1109/32.54295.

[94] R. Leveugle. Fault injection in vhdl descriptions and emulation. In *Proceedings of the 15th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*, DFT '00, pages 414–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0719-0. URL `http://dl.acm.org/citation.cfm?id=647833.738002`.

[95] R. Leveugle, R. Rochet, G. Saucier, L. Martinez, and C. Pitot. A synthesis tool for fault-tolerant finite state machines. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 502–511, June 1993. doi: 10.1109/FTCS.1993.627353.

[96] D. D. Lewis and J. Catlett. Heterogeneous uncertainty sampling for supervised learning. In *In Proceedings of the Eleventh International Conference on Machine Learning*, pages 148–156. Morgan Kaufmann, 1994.

[97] X. Li, M. C. Huang, K. Shen, and L. Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 6–6, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855840.1855846`.

[98] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. *SIGPLAN Not.*, 46 (3):213–224, Mar. 2011. ISSN 0362-1340. doi: 10.1145/1961296.1950391. URL `http://doi.acm.org/10.1145/1961296.1950391`.

[99] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 296–305, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0. doi: 10.1145/1081706.1081754. URL `http://doi.acm.org/10.1145/1081706.1081754`.

[100] LLFI. https://github.com/dependablesystemslab/llfi.

[101] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun. Classification of software behaviors for failure detection: A discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages

557–566, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-495-9. doi: 10.1145/1557019.1557083. URL `http://doi.acm.org/10.1145/1557019.1557083`.

[102] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers. SDCTune: A model for predicting the sdc proneness of an application for configurable protection. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2014 International Conference on*, pages 1–10, Oct 2014. doi: 10.1145/2656106.2656127.

[103] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman. Llfi: An intermediate code-level fault injection tool for hardware faults. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, pages 11–16, Aug 2015. doi: 10.1109/QRS.2015.13.

[104] R. Maia, L. Henriques, D. Costa, and H. Madeira. Xceptiontm - enhanced automated fault-injection environment. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 547–, 2002. doi: 10.1109/DSN.2002.1028978.

[105] M. Maniatakos, M. K. Michael, and Y. Makris. Vulnerability-based interleaving for multi-bit upset (MBU) protection in modern microprocessors. In *Test Conference (ITC), 2012 IEEE International*, pages 1–8, November 2012. doi: 10.1109/TEST.2012.6401594.

[106] M. Maniatakos, M. Michael, and Y. Makris. Investigating the limits of AVF analysis in the presence of multiple bit errors. In *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*, pages 49–54, July 2013. doi: 10.1109/IOLTS.2013.6604050.

[107] K. Z. Mao. Fast orthogonal forward selection algorithm for feature subset selection. *IEEE Transactions on Neural Networks*, 13(5):1218–1224, Sep 2002. ISSN 1045-9227. doi: 10.1109/TNN.2002.1031954.

[108] MathWorks. http://www.mathworks.co.uk/help/rtw/examples/fuel-rate-control-system.html?refresh=true.

[109] MATLAB. *version 8.3 (R2014a)*. The MathWorks Inc., Natick, Massachusetts, 2014. URL `http://www.mathworks.co.uk/products/matlab/`.

[110] D. Mease, A. J. Wyner, and A. Buja. Boosted classification trees and class probability/quantile estimation. *J. Mach. Learn. Res.*, 8:409–439, May 2007. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=1248659.1248675`.

[111] N. Miskov-Zivanov and D. Marculescu. Soft error rate analysis for sequential circuits. In *2007 Design, Automation Test in Europe Conference Exhibition*, pages 1–6, April 2007. doi: 10.1109/DATE.2007.364500.

[112] N. Miskov-Zivanov and D. Marculescu. Multiple transient faults in combinational and sequential circuits: A systematic approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29 (10):1614–1627, Oct 2010. ISSN 0278-0070. doi: 10.1109/TCAD.2010.2061131.

[113] R. Moraes, R. Barbosa, J. Duraes, N. Mendes, E. Martins, and H. Madeira. Injection of faults at component interfaces and inside the component code: are they equivalent? In *Dependable Computing Conference, 2006. EDCC '06. Sixth European*, pages 53–64, Oct 2006. doi: 10.1109/EDCC.2006.16.

[114] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones. Scalable stochastic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 335–338, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association. ISBN 978-3-9810801-6-2. URL `http://dl.acm.org/citation.cfm?id=1870926.1871008`.

[115] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira. On fault representativeness of software fault injection. *Software Engineering, IEEE Transactions on*, 39(1):80–96, Jan 2013. ISSN 0098-5589. doi: 10.1109/TSE.2011.124.

[116] K. Nguyen, J. Beugin, M. Berbineau, and M. Kassab. Modelling communication based train control system for dependability analysis of the lte communication network in train control application. In *Modelling Symposium (EMS), 2014 European*, pages 320–325, Oct 2014. doi: 10.1109/EMS.2014.55.

[117] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Application-based metrics for strategic placement of detectors. In *Dependable Computing, 2005. Proceedings. 11th Pacific Rim International Symposium on*, pages 8 pp.–, December 2005. doi: 10.1109/PRDC.2005.19.

[118] K. Pattabiraman, G. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer. Dynamic derivation of application-specific error detectors and their implementation in hardware. In *Dependable Computing Conference, 2006. EDCC '06. Sixth European*, pages 97–108, October 2006. doi: 10.1109/EDCC.2006.9.

[119] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Automated derivation of application-aware error detectors using static analysis. In *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*, pages 211–216, July 2007. doi: 10.1109/IOLTS.2007.21.

[120] M. Pazzani, C. Merz, P. Murphy, K. Ali, T. Hume, and C. Brunk. Reducing misclassification costs. In *Proc. 11th International Conference on Machine Learning*, pages 217–225. Morgan Kaufmann, 1994.

[121] S. Pemmaraju and S. Skiena. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica ®*. Cambridge University Press, New York, NY, USA, 2003. ISBN 0521806860.

[122] G. Pintér, H. Madeira, M. Vieira, I. Majzik, and A. Pataricza. A data mining approach to identify key factors in dependability experiments in dependable computing. In *Proceedings of the 5th European Dependable Computing Conference*, pages 263–280, March 2005.

[123] F. Pournaghdali, A. Rajabzadeh, and M. Ahmadi. Vhdlsfi: A simulation-based multi-bit fault injection for dependability analysis. In *Computer and Knowledge Engineering (ICCKE), 2013 3th International eConference on*, pages 354–360, Oct 2013. doi: 10.1109/ICCKE.2013.6682846.

[124] D. Powell. Failure mode assumptions and assumption coverage. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 386–395, July 1992. doi: 10.1109/FTCS.1992.243562.

[125] D. Powell, E. Martins, J. Arlat, and Y. Crouzet. Estimators for fault tolerance coverage evaluation. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 228–237, June 1993. doi: 10.1109/FTCS.1993.627326.

[126] R. T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *AFIPS Joint Computer Conferences*, pages 133–138, 1959.

[127] J. R. Quinlan. Learning logical definitions from relations. *Mach. Learn.*, 5 (3):239–266, Sept. 1990. ISSN 0885-6125. doi: 10.1023/A:1022699322624. URL `http://dx.doi.org/10.1023/A:1022699322624`.

[128] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1-55860-238-0.

[129] J. R. Quinlan and R. M. Cameron-Jones. *FOIL: A midterm report*, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993. ISBN 978-3-540-47597-2. doi: 10.1007/3-540-56602-3_124. URL `http://dx.doi.org/10.1007/3-540-56602-3_124`.

[130] D. Radaelli, H. Puchner, S. Wong, and S. Daniel. Investigation of multi-bit upsets in a 150 nm technology sram device. *IEEE Transactions on Nuclear Science*, 52(6):2433–2437, Dec 2005. ISSN 0018-9499. doi: 10.1109/TNS.2005.860675.

[131] P. M. B. Rao, M. Ebrahimi, R. Seyyedi, and M. B. Tahoori. Protecting SRAM-based FPGAs against multiple bit upsets using erasure codes. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 212:1–212:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2730-5. doi: 10.1145/2593069.2593191. URL `http://doi.acm.org/10.1145/2593069.2593191`.

[132] R. A. Reed, M. A. Carts, P. W. Marshall, C. J. Marshall, O. Musseau, P. J. McNulty, D. R. Roth, S. Buchner, J. Melinger, and T. Corbiere. Heavy ion and proton-induced single event multiple upset. *IEEE Transactions on Nuclear Science*, 44:2224–2229, Dec. 1997. doi: 10.1109/23.659039.

[133] R. Reicherdt and S. Glesner. Slicing MATLAB simulink models. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 551–561, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL `http://dl.acm.org/citation.cfm?id=2337223.2337288`.

[134] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: software implemented fault tolerance. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 243–254, March 2005. doi: 10.1109/CGO.2005.34.

[135] C. Rousselle, M. Pflanz, A. Behling, T. Mohaupt, and H. T. Vierhaus. A register-transfer-level fault simulator for permanent and transient faults in embedded processors. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pages 811–, 2001. doi: 10.1109/DATE.2001.915148.

[136] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. V.Adve, V. S. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 70–79, June 2008. doi: 10.1109/DSN.2008.4630072.

[137] F. Salles, M. Rodriguez, J.-C. Fabre, and J. Arlat. Metakernels and fault containment wrappers. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, pages 22–29, November 1999.

[138] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993518. URL `http://doi.acm.org/10.1145/1993498.1993518`.

[139] B. Sangchoolie, F. Ayatolahi, R. Johansson, and J. Karlsson. A study of the impact of bit-flip errors on programs compiled with different optimization levels. In *Dependable Computing Conference (EDCC), 2014 Tenth European*, pages 146–157, May 2014. doi: 10.1109/EDCC.2014.30.

[140] M. Santos, R. Martins, M. Santana, and S. Fernandes. An adaptive random heuristic in virtual networks: Dependability analysis. In *Network Operations and Management Symposium (LANOMS), 2015 Latin American*, pages 41–48, Oct 2015. doi: 10.1109/LANOMS.2015.7332668.

[141] S. Satoh, Y. Tosaka, and S. Wender. Geometric effect of multiple-bit soft errors induced by cosmic ray neutrons on DRAMs. *Electron Device Letters, IEEE*, 21(6):310–312, June 2000. ISSN 0741-3106. doi: 10.1109/55.843160.

[142] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 38–48, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: 10.1145/268946.268950. URL `http://doi.acm.org/10.1145/268946.268950`.

[143] B. Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin, Madison, 2009.

[144] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel. Exploiting program-level masking and error propagation for constrained reliability optimization. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–9, May 2013.

[145] V. Sieh, O. Tschache, and F. Balbach. Verify: evaluation of reliability using vhdl-models with embedded fault descriptions. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 32–36, June 1997. doi: 10.1109/FTCS.1997.614074.

[146] M. Sipser. *Introduction to the Theory of Computation.* International Thomson Publishing, 1st edition, 1996. ISBN 053494728X.

[147] M. Sipser. *Introduction to the theory of computation.* Thomson Course Technology, Boston, 2006. ISBN 0-534-95097-3. URL `http://opac.inria.fr/record=b1119335`.

[148] D. Skarin and J. Karlsson. Software implemented detection and recovery of soft errors in a brake-by-wire system. In *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*, pages 145–154, May 2008. doi: 10.1109/EDCC-7.2008.24.

[149] D. Skarin, R. Barbosa, and J. Karlsson. Goofi-2: A tool for experimental dependability assessment. In *2010 IEEE/IFIP International Conference*

on Dependable Systems Networks (DSN), pages 557–562, June 2010. doi: 10.1109/DSN.2010.5544265.

[150] V. Sridharan and D. Liberty. A study of DRAM failures in the field. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, Nov 2012. doi: 10.1109/SC.2012.13.

[151] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer. Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International*, pages 91–100, 2000. doi: 10.1109/IPDS.2000.839467.

[152] SUSAN. http://users.fmrib.ox.ac.uk/∼steve/susan/susan2l.c.

[153] K. Suyama and N. Sebe. Dependability analysis of fault-tolerant servo systems using limited integrators. In *Control Conference (ECC), 2014 European*, pages 652–659, June 2014. doi: 10.1109/ECC.2014.6862235.

[154] D. D. Thaker, D. Franklin, J. Oliver, S. Biswas, D. Lockhart, T. Metodi, and F. T. Chong. Characterization of error-tolerant applications when protecting control data. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 142–149, Oct 2006. doi: 10.1109/IISWC.2006.302738.

[155] A. Thomas and K. Pattabiraman. Error detector placement for soft computation. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12, June 2013. doi: 10.1109/DSN.2013.6575353.

[156] A. Thomas and K. Pattabiraman. LLFI: An intermediate code level fault injector for soft computing applications. In *Proceedings of IEEE Workshop on Silicon Errors in Logic, System Effects (SELSE)*, 2013.

[157] K. M. Ting. An instance-weighting method to induce cost-sensitive trees. *IEEE Trans. on Knowl. and Data Eng.*, 14(3):659–665, May 2002. ISSN 1041-4347. doi: 10.1109/TKDE.2002.1000348. URL `http://dx.doi.org/10.1109/TKDE.2002.1000348`.

[158] E. Touloupis, J. A. Flint, V. A. Chouliaras, and D. D. Ward. Study of the effects of SEU-induced faults on a pipeline protected microprocessor. *Computers, IEEE Transactions on*, 56(12):1585–1596, December 2007. ISSN 0018-9340. doi: 10.1109/TC.2007.70766.

[159] J. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson:. Reducing critical failures for control algorithms using executable assertions and best effort recover. In *Proceedings Dependable Systems and Networks*, pages 347–356, 2001.

[160] L. Wang, Z. Kalbarczyk, and R. K. Iyer. Formalizing system behavior for evaluating a system hang detector. In *Reliable Distributed Systems, 2008. SRDS '08. IEEE Symposium on*, pages 269–278, October 2008. doi: 10.1109/SRDS.2008.11.

[161] J. Wei, L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. Comparing the effects of intermittent and transient hardware faults on programs. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 53–58, June 2011. doi: 10.1109/DSNW.2011.5958835.

[162] M. Wilkening, V. Sridharan, S. Li, F. Previlon, S. Gurumurthi, and D. R. Kaeli. Calculating architectural vulnerability factors for spatial multi-bit transient faults. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 293–305, December 2014. doi: 10.1109/MICRO.2014.15.

[163] S. Winter, M. Tretter, B. Sattler, and N. Suri. simFI: From single to simultaneous software fault injections. In *Dependable Systems and Net-*

works (DSN), 2013 43rd Annual IEEE/IFIP International Conference on, pages 1–12, June 2013. doi: 10.1109/DSN.2013.6575310.

[164] X. Xu, W. Chen, J. Wan, and R. Yu. Distributed fault diagnosis of wireless sensor networks. In *Communication Technology, 2008. ICCT 2008. 11th IEEE International Conference on*, pages 148–151, Nov 2008. doi: 10.1109/ICCT.2008.4716155.

[165] C. Yang and A. Orailoglu. Processor reliability enhancement through compiler-directed register file peak temperature reductionprocessor reliability enhancement through compiler-directed register file peak temperature reduction. In *Proceedings Dependable Systems and Networks*, pages 468–477, 2009.

[166] B. Zadrozny, J. Langford, and N. Abe. Cost-sensitive learning by cost-proportionate example weighting. In *Proceedings of the Third IEEE International Conference on Data Mining*, ICDM '03, pages 435–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1978-4. URL `http://dl.acm.org/citation.cfm?id=951949.952181`.

[167] Y. Zennir and R. Bendib. Modeling and dependability analysis of an industrial plant: Case study. In *Industrial Engineering and Systems Management (IESM), 2015 International Conference on*, pages 1012–1018, Oct 2015. doi: 10.1109/IESM.2015.7380278.

[168] F. Zhao, H. Jin, D. Zou, and P. Qin. Dependability analysis for fault-tolerant computer systems using dynamic fault graphs. *China Communications*, 11(9):16–30, Sept 2014. ISSN 1673-5447. doi: 10.1109/CC.2014.6969708.