A. D. MCCXXIV

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Comunità Europea
Fondo Sociale Europeo

ON THE QUALITY OF FAULT INJECTION FOR OFF-THE-SHELF COMPONENTS IN SAFETY-CRITICAL SYSTEMS

ANNA LANZARO

Tesi di Dottorato di Ricerca
(XXVI Ciclo)
Marzo 2014

Il Tutore
Prof. Domenico Cotroneo

Il Coordinatore del Dottorato
Prof. Francesco Garofalo

Dipartimento di Ingegneria Elettrica
e delleTecnologie dell’Informazione

ON THE QUALITY OF FAULT INJECTION FOR OFF-THE-SHELF COMPONENTS IN SAFETY-CRITICAL SYSTEMS

By
Anna Lanzaro

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II
VIA CLAUDIO 21, 80125, NAPLES, ITALY
MAY 2014

Acknowledgements

First, I would like to thank my advisor prof. Domenico Cotroneo for his professional support, for his constantly encouragement and, above all, for his trust in me.

A special thanks goes to Roberto Natella, who was a great co-advisor during my master degree and he continued to patiently support me during these three years. I am really glad and I feel honoured to work with him.

I also want to thank Antonio Pecchia, who gave me important and valuable advises during my PhD. He is my colleague, but he is especially a good friend.

Concluding, I would thank both MOBILAB and CINI groups, including people who were in our lab since the beginning of my experience and the ones who have just arrived. I wish them all the best.

Anna

Table of Contents

List of Tables **vii**

List of Figures **viii**

1 Introduction **1**

1.1 Thesis contributions . . . 7

2 Off-The-Shelf Components in Safety-Critical Systems **12**

2.1 Introduction . . . 12

2.2 Off-The-Shelf Components: Definitions and Classification . . . 13

2.3 OTS Components and Safety Standards . . . 14

2.4 Testing OTS-based Systems . . . 17

2.4.1 Dependability: Basic Concepts . . . 18

2.4.2 Fault Injection Testing . . . 19

3 Achieving Accuracy in Binary Code Mutation **23**

3.1 Introduction . . . 23

3.2 Background and Related Work . . . 25

3.2.1 G-SWFIT . . . 29

3.2.2 SAFE . . . 30

3.3 Experimental Evaluation of Binary Fault Injection . . . 32

3.3.1 Fault Matching . . . 35

3.3.2 Fault Sampling . . . 37

3.3.3 Case Study 41
3.3.4 Results 42
3.4 Systematic Testing of Binary Fault Injection 54
3.4.1 Test-suite Generation 63
3.4.2 Test-suite Execution, Comparison and Detection of Inaccuracies 67
3.4.3 The csXception^{TM} suite 71
3.4.4 Test Planning 73
3.4.5 Results 73
3.5 Summary 79

4 Achieving Representativeness in Interface Error Injection 84

4.1 Introduction 84
4.2 Background and Related Work 86
4.3 Propagation of Errors at Component Interfaces 92
4.3.1 Propagation analysis approach 93
4.3.2 Component fault injection 105
4.3.3 Results 107
4.4 Summary 114

5 Software-Implemented Fault Injection in the Multicore Era 117

5.1 Introduction 117
5.2 Multicore in safety-critical systems 119
5.3 Background and Related Work 122
5.3.1 Software-implemented Error Injection for Multicore 126
5.3.2 Case Study 128
5.3.3 Campaign #1 133
5.3.4 Campaign #2 135
5.4 Emulating Hardware Errors in Virtualized Systems 136
5.4.1 Case study 138
5.4.2 Campaign 1 141
5.4.3 Campaign 2 142

5.5 Summary 143

6 Conclusions and Future Work **144**

Bibliography **150**

List of Tables

3.1 Fault Types (see also [36]). 26

3.2 Classification of Fault Injection Tools 29

3.3 Description of OMFC Fault Type 30

3.4 Comparison of Average Software Complexity Metrics of Functions in RTEMS and CDMS Code 48

3.5 Fault Types of G-SWFIT [36]. 55

3.6 Constraints of Fault Types in G-SWFIT [36] 56

3.7 Parameters of the *Faultprog* random program generator. 66

3.8 Test-suites generated by FaultProg 74

4.1 Fault types adopted in this study [36]. 106

4.2 Outcomes of experiments. 109

4.3 Distributions of return values in fault injection experiments. 113

4.4 Correlation between corruption rate and number of accesses. 114

5.1 Status Register [15:0] 131

5.2 MCEs injector input 139

5.3 MCE example 140

List of Figures

2.1 Component and Interface 13

2.2 General fault injection framework 20

3.1 Software Fault Injection Techniques 32

3.2 Overview of the Method for G-SWFIT Evaluation 35

3.3 Fault Matching Procedure 37

3.4 Examples of Spurious and Omitted Faults Due to C Preprocessor Macro . . . 39

3.5 Architecture of the Case Study 42

3.6 Distributions of Software Faults at both Binary and Source Code Level 43

3.7 Correctly Injected, Spurious, and Omitted Faults 45

3.8 Causes of Incorrect Fault Injection in the Case Study 46

3.9 Number of Faults (Correctly Injected, Spurious, and Omitted) in OS and Application Code 46

3.10 Causes of Incorrect Fault Injection in OS and Application code 48

3.11 Spurious MFC Fault in CDMS. 51

3.12 Omitted MFC Fault in CDMS 52

3.13 Omitted MIA Fault in CDMS. 53

3.14 Number of Faults (Correctly Injected, Spurious, and Omitted) when Fixing Implementation Issues of the G-SWFIT Tool 54

3.15 Accuracy of G-SWFIT in the Context of an Embedded Space Software [29]. . 58

3.16 General Structure of a Synthetic Program 60

3.17 Proposed Approach 61

3.18 Example of a Synthetic Program for Testing MFC Fault Type 67

3.19 csXception^{TM} architecture 72

3.20 Distributions of Faults Injected at Binary and Source Code Level 75
3.21 Distributions of Correctly Injected, Spurious and Omitted Faults. 75
3.22 Spurious injections. 78
3.23 Example of synthetic program causing a spurious injection. 79
3.24 Omitted injections. 80
3.25 Example of synthetic program causing an omitted injection. 81

4.1 Relationship between component faults and interface errors. 90
4.2 Propagation through a library-allocated heap area. 91
4.3 Propagation through a user-allocated heap area. 92
4.4 Propagation through a library-allocated heap area, reached through a user-allocated heap area. 93
4.5 Propagation through a user-allocated local variable. 94
4.6 Overview of error propagation analysis. 95
4.7 Example of reachability graph. 98
4.8 Trace pre-processing. 100
4.9 Example of comparison between faulty and fault-free traces. 102
4.10 Software Fault Injection approach [28, 80]. 106
4.11 Cumulative distribution (per library) of the number of corrupted bytes of interface data. 111
4.12 Cumulative distribution (per fault type) of the number of corrupted bytes of interface data. 112
4.13 Byte corruption rate and number of accesses for *Libxml2*. 115

5.1 Proposed Error Injection Framework 127
5.2 Intel Core i7 Architectural Block Diagram 129
5.3 Machine Check Architecture 130
5.4 MCE Description File and Severity Levels 132
5.5 SER_P=0: Recovery Actions Not Supported by the Processor 134
5.6 SER_P=1: Recovery Actions Supported by the Processor 134
5.7 Severity and Recovery Actions Grouped by Error Categories 136
5.8 Severity and Recovery Actions for Campaign #2 137

5.9 Injection Framework for Virtualized Systems 137
5.10 Architecture Framework for Xen 139
5.11 MCE Generator 141
5.12 Recovery Actions for Campaign #2 142
5.13 SER_P=1: Recovery Actions Implemented by Xen 143

Chapter 1

Introduction

Fault Injection (FI) is a family of techniques that emulates hardware and/or software faults by deliberately inserting them into a system component in order to analyze system behavior under faulty conditions, i.e. whether the system can tolerate faults. It is well recognized that Fault Injection is a powerful means for dependability assessment, especially when the system is composed by third-party components, which are often distributed as executables (commercial Off-The-Shelf), being their **source-code not available**.

The integration of hardware and software OTS components is quite common in the development of modern systems. Their adoption is driven by development costs, market pressure, and performance reasons. Similar reasons are behind the adoption of OTS components in safety-critical systems where standards e.g., ISO-26262 [56], IEC-61508 [27], and DO-178B [42] regulate their integration by requiring evidences that OTS components fulfil safety functions. Unfortunately, the lack of documentation and information about the development life-cycle and different operational environments often lead to an improper integration of OTS components, rising the risk of failures [116]. As a consequence, dependability assessment techniques and proper verification strategies are mandatory in order to evaluate system behavior in presence of hardware and software faults.

Regarding the emulation of software faults, fault injection is able to operate at different system levels: at component-level by means of *code mutation* techniques; and at interface-level through *error injection* techniques. Code mutation techniques emulate bugs into a system by mutating the code of a program. Mutations represent software defects, i.e. programming mistakes, to be introduced in the code in order to realistically emulate a faulty software . In the case of OTS-based systems, the injection of programming errors is performed at at binary-level and it presumes that programming constructs used in the source code are identified by looking only at the binary code. Unfortunately, software fault injection (SFI) at binary-level is a difficult and error-prone task due to the complexity of programming languages and of modern compilers, which make difficult and in some cases impossible to accurately recognize where to inject faults. The major concern when injecting software faults at binary-level is to assure that binary-level mutations are accurately performed to emulate programming mistakes, i.e. **SFI has to correctly emulate software faults to an acceptable degree of confidence that a fault injected in the binary**

code correctly emulates a software defect in the source code. Inaccuracies in the injection could negatively affect the results of fault injection campaigns leading to erroneously considerations on dependability properties of the system.

Instead of injecting software faults, *Interface Error Injection* (IEI), which is often adopted in the context of *robustness testing*, mimics the effects (i.e., errors) produced by faults in a component, by *injecting exceptional or invalid values* at the component's interface [48, 73, 118]. Despite its popularity, the use of IEI for the representative emulation of component faults (as required by dependability assessment strategies [62,78,115]) is questionable. There are not evidences that **the injection of interface errors can realistically emulate faults occurring in the system**. Investigations on the representativeness of interface errors is required in order to perform an effective and representative error injection into the component's interface.

Regarding the emulation of hardware faults, a great variety of fault injection techniques exists. One of the most popular hardware fault injection techniques is known as *software-implemented* fault injection, i.e. SWIFI. Based on bit stuck-at and bit-flip models, SWIFI allows the emulation of hardware faults through software by reproducing possible effects of real hardware errors without directly interfering with the system. Although these techniques were successfully employed for the evaluation of system behavior against hardware errors, they could result to be not effective in the case of advanced processors, such as multicore ones. Multicore integrates more cores on the same die that, running in parallel, share many resources (e.g. memory, caches, registers). The complexity of the architecture increases the probability of hardware errors because of the great number of transistors on the same chip. To make processors more reliable, designers and developers have devised many hardware-implemented mechanisms such as error detection that are able to detect and, in some case, to correct hardware errors. Detected errors are then reported to the upper software layer, i.e. operating system. So that, the complexity is shifted to the software that has to correctly interpret signaled hardware errors and to implement adequate software recovery mechanisms to cope with them. Although SWIFI techniques can be applied also when system integrates multicore processors, **fault injection campaign could be very expensive and unfeasible** due to the huge number of resources, i.e. possible locations where to inject errors. In fact, the replication of cores means also replicated resources per-core. Moreover, two aspects should be considered when adopting existing fault models. First, single or multiple bit-flip could be automatically corrected by hardware mechanisms and masked to the software **affecting the effectiveness of SWIFI techniques**. Second, new errors (that were not a concern in single-core architectures) may occur, e.g. errors in the interconnection links between cores make **existing SWIFI techniques not representative enough for modern systems**. Effective fault injection techniques should evaluate the behavior of software systems deployed on the top of multicore. This is very important also in safety-critical systems where the increasing trend to integrate hardware Off-the-Shelf components is driven by the need of deliver sophisticated and demanding functionalities by assuring innovative solutions, high performance and reduced costs.

Software-based systems in the automotive domain are an example. Modern cars includes many controllers, sensors and actuators connected by different bus for adaptive cruise control with "stop and go" capabilities, stability control, brake assistance or park assistance, etc. To meet the requirements of the modern functionalities, higher performance are demanded, but building ad-hoc advanced hardware components could be very expensive. Exploiting existing components and, above all, taking advantages from technological progress, the integration of hardware Off-the-shelf components seem to be a good choice. Historically, critical embedded systems have been developed using hardware components based on single-core processors, but nowadays they cannot overlook the advances of the micro-controllers market from which general purpose systems such as personal computers, smart-phones and tablet already have beneficed: they have undergone a significant change by replacing single-core processors with multi-core. The migration brings many benefits in terms of performance, power consumption and energy efficiency although it is an awkward activity in critical domains that have to accomplish some strict standards. To exploit the full potential of multi-core, some changes in software are required: tasks should be executed in parallel, i.e. thread level parallelism, to benefit from multi-core. In addition, software running on the top of multi-core has to take into account architectural changes and improvements: more cores on board connected by links, shared resources, several levels of cache, advanced error handling mechanisms. Unfortunately, we are still far away from the use of parallel programming in critical domains because of problems related to the non-determinism of the execution, to the synchronization and the cooperation between tasks running in parallel.

Nevertheless, multi-core, in conjunction with virtualization [44], can support achieving safety requirements imposed by standards. Virtualization makes it possible running independent applications on each core to ensure properties such as, space and temporal isolation. Furthermore, the inherent presence of replicated cores allows implementing fault-tolerant solutions. Overall these features exacerbate the need for strategies to evaluate dependability characteristics of multi-core systems.

In conclusion, fault injection is widely adopted in the verification and validation of OTS-based systems. Effective FI techniques should guarantee a certain level of **quality**, i.e they must guarantee *representativeness* and *accuracy* properties: **FI techniques should inject in an accurate way faults that are representative.** If these features are not guaranteed, analyses based on fault injection experiments can result to be wrong and, in worst cases, they may contribute to dramatic accidents and economic loss.

1.1 Thesis contributions

Considering code mutation, interface error injection and software-implemented fault injection, the main research questions are:

- SFI: Are mutation at binary level accurate enough? How is it possible to validate the correctness of the modifications into the binary code, being sure that injections are correctly performed?
- IEI: Are the existing error model representative? Is an error injected at component interface representative for a real software bug in a component?
- SWIFI: How emulate hardware errors in multicore processors? Are the existing operating systems and hypervisors able to correctly treat errors signalled by hardware error detection and reporting mechanisms of modern processor such as multicore?

This dissertation contributes to the improvement of the quality, i.e. representativeness and accuracy properties, of Fault Injection techniques employed in the evaluation of COTS-based safety-critical systems by proposing:

1. **A method for testing and improving the accuracy of BCM tools**. Software fault injection based on code mutation at binary level requires that programming constructs used in the source code are identified by looking only at the binary code, since the injection is performed at this level. The proposed method is based on the

automatic generation of *synthetic programs* that are given as inputs to a BCM tool in order to evaluate its accuracy at performing binary mutations. First, several synthetic programs are generated by encompassing different programming constructs in different contexts (e.g., nested loops, control flow constructs and function calls). Then, the BCM tool is applied on the binary code of the synthetic programs, and the mutations produced by the BCM tool are compared against the source code of the synthetic programs (the analysis of source code serves as a reference, as it does not suffer the limitations of binary level mutation), in order to assess the **ability of the BCM tool to correctly recognize and mutate programming constructs at the binary level**, and to reveal its issues and limitations. In other words, the set of synthetic programs acts as a test suite for evaluating and improving binary-level fault injection and mutation testing tools.

2. **A method for analysing error propagation at the interfaces of software components.** The method aims at automatically analyzing how software faults in components' code result in errors at components' interfaces, in order to provide some constructive evidence towards more representative IEI techniques. It identifies how faults in software components manifest as interface errors. First, faults are injected in the software component under analysis by using a fault injection technique. Then, it instruments and executes the software component and identifies the effects of injected

faults on the program that uses the component, including the corruption of data structures shared between the program and the component and erroneous return values from function calls.

3. **A prototype framework for injecting hardware errors in multicore-based architecture** The tool aims at injecting hardware errors by exploiting the error reporting architecture implemented in modern processors as multicore in order to asses error handling mechanisms of existing operating systems. Fault injection campaigns have been conducted to test the functionalities of the framework under the Linux OS running on the top of the Intel i7 processor. Based on the same approach, a prototype tool was also proposed to inject errors in virtualized system in order to validate the error handler mechanisms implemented in the hypervisors.

This thesis includes materials from the following research papers, already published in peer-reviewed conferences and journals or submitted for review:

- *Experimental Analysis of Binary-Level Software Fault Injection in Complex Softwar*e, D. Cotroneo, **A.Lanzaro**, R. Natella,R. Barbosa, Proc. of 9th European Dependable Computing Conference (EDCC), May 2012, Sibiu, Romania. **(Best Presentation Award)**

- *Multicore Systems: Challenges for creating a representative fault model for fault injection*, N. Silva, R. Barbosa, **A. Lanzaro**, D. Cotroneo, J. Duraes, DASIA International Space System Engineering Conference, 2012.

- *Injecting Machine Check Errors to Explore Dependability Issues of Multicore Systems*, **A. Lanzaro**, A. Pecchia, M. Cinque, D. Cotroneo, N.Silva, R.Barbosa, Supplemental Volume of the Proc. of 42nd International Conference on Dependable Systems and Networks(DSN), 2012.

- *A Preliminary Fault Injection Framework for Evaluating Multicore Systems*, **A. Lanzaro**, A. Pecchia, M. Cinque, D. Cotroneo, R. Barbosa, and N. Silva, Supplemental volume of Proc. of 32nd International Conference on Computer Safety, Reliability and Security (SAFECOMP), 2012.

- *Tools for Injecting Software Faults at the Binary and Source-Code Level* **A.Lanzaro**, R.Natella, R.Barbosa, Innovative Technologies for Dependable OTS-Based Critical Systems - Challenges and Achievements of the CRITICAL STEP Project (2013)

- *Leveraging Fault Injection Techniques in Critical Industrial Applications* A.Pecchia, **A.Lanzaro**, As'ad Salkham, M.Cinque, N.Silva, Innovative Technologies for Dependable OTS-Based Critical Systems - Challenges and Achievements of the CRITICAL STEP Project (2013)

- *An Empirical Study of Injected versus Actual Interface Errors*, **A. Lanzaro**, R.Natella, S.Winter, D.Cotroneo, N.Suri. Procs of International Symposium on Software Testing and Analysis (ISSTA), 2014

Part of the activities were conducted in collaboration with Critical Software (Coimbra, PT) in the context of the European Project Critical-step "Marie-Curie" Industry-Academia Partnerships and Pathways (IAPP) FP7-PEOPLE-2008-IAPP [2].

Chapter 2

Off-The-Shelf Components in Safety-Critical Systems

2.1 Introduction

Economical reasons along with technological advances influence the development of systems: the main challenge for developers is to build complex and performing systems by having reduced resources. Traditionally, software-based systems were developed from scratch, but nowadays it is not thinkable because of the high number of functionalities demanded to software. One of the strategy to reduce development costs/time and to exploit new technologies is the integration of components developed by third-party, i.e Off-the-shelf components. In literature, there are many studies that discuss issues related to the selection, integration, maintenance and certification of OTS components in critical systems [69], but they out of the scope of the thesis.

Instead, this dissertation is focused on dependability evaluation of OTS-based systems through adequate and effective testing techniques once OTS components are selected and integrated into safety-critical systems. In particular, this chapter describes the use of OTS components in safety-critical systems by highlighting suggestions provided by the standards about OTS components and testing techniques.

2.2 Off-The-Shelf Components: Definitions and Classification

A *component* is an independent and reusable unit whose integration in a system contributes to its functioning. It communicates with the other parts of the system by means of one or more *interfaces.* Interface is an access point by which components provide services that can be required from clients, i.e. other components. For example, in Figure 2.1 component C1 provides the service S. Component C2 is the client that can require the service S by means of its interface.

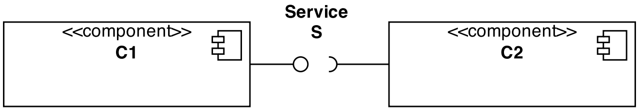


Figure 2.1: Component and Interface

In hardware engineering, the concept of Off-the-shelf component is well understood (e.g. RAM, CPU, etc.) and and its integration to build more complex systems is a common practice, namely Intellectual property (IP) based development. Instead, for what concern software, an OTS component is not well defined. In literature, there are many and confusing definitions [107], [86], [113], [19], [16] and the terms Off-the-Shelf (OTS) and Commercial Off-the-Shelf (COTS) are often improperly used. Moreover, depending on the domain, OTS components are also called Government-Off-The-Shelf (GOTS), Military-Off-The-Shelf (MOTS) and Non-Developmental Item (NDI) when components are not-commercially acquired; Open Source Software (OSS) when the source code is available but not modifiable [76]; Software of Unknown Pedigree (SOUP) [18] used in medical domain.

However, according with [107] and [86], we adopt following definitions:

- An *OTS component* is acquired by third-party and it is usually distributed as an executable, so the source code is not available. The latter, when available, cannot be modified.

- A *COTS component* is an OTS component that (i) is developed by a commercial vendor; (ii) it is available to the general public; (iii) it can be bought (or leased, or licensed) [86].

- An *Open source software* is an OTS component that uses open specifications for interfaces, services, and supporting formats.

2.3 OTS Components and Safety Standards

The advantages provided by the OTS integration in critical domains were recognized since 1994 when the US Department of Defence observed that the technology used military systems was 10 years old while new technology emerges every 18-24 months [86]. Since that moment, military systems started the migration to software developed using COTS components. The Department of Defense stated that "to meet future needs, the Department of Defense must increase access to commercial state-of-the-art technology" and "moving to greater use of performance and commercial specifications and standards is one of the most important actions that DoD must take to ensure we are able to meet our military, economic, and policy objectives in the future" [89].

Many recent standards [38] provide guidance on using both hardware and software COTS components as discussed in [71]:

- **IEC 61508** [27] is the standard for Electrical/Electronic/Programmable Electronic. It states that the integration of OTS components can bring many advantages, but adequate verification and validation process should provide evidences that components meet safety requirement. Negative effects due to different operational environment, functionalities not required in the new context but implemented in the component, and internal operations of the OTS component not being fully understood should be taken into account during V&V process. It suggests methods such as interface testing, error guessing and seeding (i.e. fault/error injection), and functional testing under environmental conditions.

- **MIL-STD-882D** [32] is the safety standard adopted in US military and defense domains. It provides suggestions for software development and it requires a safety program to identify hazards and prevent mishaps. Moreover, it recommends additional hazard analyses in case of OTS components by using unit, integration and qualification testing.

- **Def Stan 00-55 and 00-56** [101] [102] are safety standards adopted for the development of UK military and defense systems. They suggest system or component evaluation to determine whether they satisfy the requirements. FMEA is used to perform process failure analysis. In particular, Def Stan 00-56 states: "Where COTS or other existing complex electronic elements are used, the Safety Case should detail the processes used for evaluation, validation and implementation of the complex electronic element, the processes used for any bespoke software or hardware (such as software wrappers or hardware interlocks) and any information from the complex electronic element supplier about the development process (where available). In general, the more onerous the safety integrity requirements, the more rigorous and compelling the process evidence that should be provided. For a COTS or pre-existing element, the rigour may have to be provided at the evaluation stage."

- **DO-178B** [42] is the standard for aerospace domain. It concerns software employed in airborne systems. Both hardware and software components are considered and studied during safety assessment and verification activities. It suggests the analysis of historical data (e.g. data about the reliability) when adopting existing software in new aircraft and software whose data does not satisfy the guidelines of the standard in order to provide evidences and justify the integration.

2.4 Testing OTS-based Systems

Safety standards require evidences that systems are dependable even in presence of OTS/-COTS components, i.e they should tolerate faults. In particular, they suggest the implementation of fault tolerant mechanisms in order to avoid system failures. To accomplish this, these mechanisms have to detect faults and errors that occur during system functioning and perform recovery actions to restore system state. Approach used are based of replication and diversity: examples are Recovery blocks, N-versioning programming.

However, to asses the effectiveness of fault tolerant mechanisms, to evaluate error handlers and error propagations among components, adequate and effective testing techniques are required.

As discussed in 2.2, an OTS components is usually distributed as an executable, i.e. the source code is not available. Due to the nature of OTS components, fault injection techniques seems to be suitable for the aim.

2.4.1 Dependability: Basic Concepts

The fault-error-failure chain expresses the concept that the activation of a fault leads to an error. An invalid state generated by the error may lead to another error or to a failure [14]. In particular:

- **Fault** is a defect in a system. It can affect software components, e.g. software bugs, and hardware components, e.g. physical faults. If activated, it causes an error.
- **Error** is the deviation of the system states from a correct state that may lead to a subsequent (service) failure. Errors are dormant if they do not cause service failure, it they manifest at system or component interface, a failure occurs. Errors can propagate from one component to another of the system through their interfaces.
- **Failures** or service failure is the deviation of the system from the correct implementation of the system function. A failure may occur because the system violates the specification or because the specification is not adequate to describe the behavior of the system. The failure of one or more services implementing system functions make the system operate in a degraded mode.

A system that can avoid service failures that are more frequent and more severe than is acceptable is called *dependable system*. Dependability encompass a set of attributes:

- Availability: readiness for correct service.

- Reliability: continuity of correct service.
- Safety: absence of catastrophic consequences on the user(s) and the environment.
- Integrity: absence of improper system alterations.
- Maintainability: ability to undergo modifications and repairs.

2.4.2 Fault Injection Testing

Fault Injection is a technique that emulates hardware and software faults by deliberately inserting defects into a system component in order to determine how the system behaves when a component fails, i.e. whether the system can tolerate faults. Due to its ability of emulating a malfunctioning in component of target system, FI is widely is considered a valuable dependability assessment approach. It was successfully used for **validating fault-tolerance mechanisms** by evaluating how a target behaves in presence of faulty components through error detection and handling mechanisms (such as assertions and exception handlers) against component faults [11,67,85]. In [53,78,114], FI techniques were adopted for **aiding FMECAs (Failure Mode, Effects, and Criticality Analysis)**: Developers can quantify the impact of a faulty component on the overall system (e.g., in terms of catastrophic system failures), and mitigate risks by comprehensively testing the most critical components and revising the system design. Moreover, FI can helps in the context of **Dependability benchmarking** by providing support for developers to choose among alternative systems or components the one that provides the best dependability and/or performance in the presence of other, faulty, components [62].

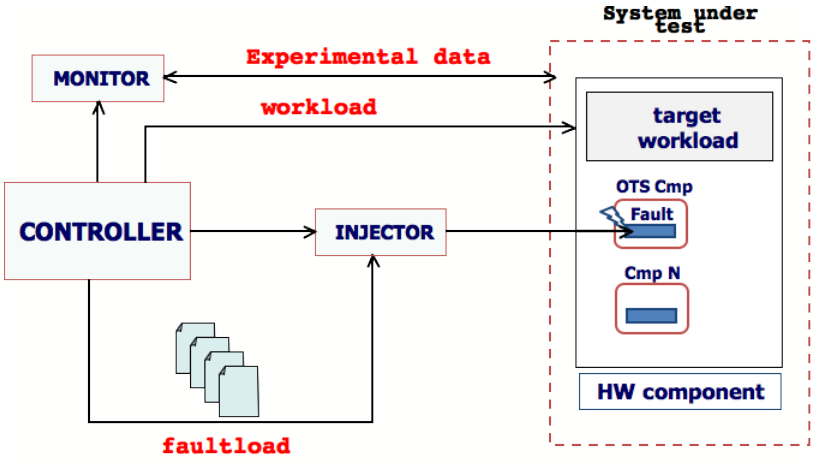


Figure 2.2: General fault injection framework

Figure 2.2 depicts a general framework for fault injection. In the following list, key elements are discussed highlighting the characteristics they should have in order to meet the requirements for an effective, efficient and good quality fault injection.

- **The target** is the system under test that executes a **workload**, an operating system, an application or a program. A workload should be *representative* for a real system utilization.

- **Faultload** is the set of faults that will be injected in the target system during the campaign; faults to be injected are defined by a *fault model* describes faults in terms of their type (what to inject), location (where to inject) and trigger (when to inject). It has to be representative, i.e. a fault has to represent a real defect. *Representativeness* of faultloads is achieved by defining a realistic fault model, instead the *accuracy* is achieved by reproducing this fault model when faults are injected.

- **Injector** is the component responsible of introducing fault into the target. The injector should not distort the actual behaviour of the system under test. The isolation between the injector and the target should be achieved to satisfy the *not-intrusiveness* property.

- **Monitor** is responsible for collecting data concerning the fault-injection outcomes. Monitor should cope with data loss caused by experiments leading to critical system failures, such as reboot or panic. Again, monitoring and data collection features should not impact the behaviour of the target system. This contributes to *controllability* and *observability* properties.

- **Controller** is the entity responsible for iterating fault injection experiments and coordinating the described components. For each experiment it activates/deactivates the injector module, and stores monitoring data. Moreover, controller should ensure that the workload is actually running at the time injection is performed. It has to fulfil controllability and *repeatability* properties.

This dissertation is focused on the quality of fault injection techniques represented by *representativeness* and *accuracy* essential properties for meeting quality requirements. The aim is to evaluate and to improve the quality of existing fault techniques for the assessment of OTS-based systems in safety-critical contexts.

In origin, fault injection techniques were able to emulate hardware faults by physically interfering with the target system through special and expensive devices. To overcome limitations of such techniques, software-implemented fault injection techniques were proposed. They injects hardware faults by emulating the effects of faults (e.g., CPU or memory faults), i.e. corrupting the state of the software using bit-flipping or stuck-at techniques. these techniques are also known as error injection. More recently, software injection techniques emerged. At component level, injection is performed by mutating the source code or the binary code of a component of the target system. Instead, at interface level, software faults are inserted into the component interface of the target.

Chapter 3

Achieving Accuracy in Binary Code Mutation

3.1 Introduction

Software Fault Injection aims at the realistic emulation of *software faults* (i.e., bugs[1]) in a software component to assess the impact of these faults on the system behavior. SFI is assuming an increasing relevance since software faults have been recognized as one of the major causes of system failures [49, 87]. It is used for the experimental validation and improvement of fault tolerance mechanisms and algorithms [10, 85]; it makes possible to analyze worst-case scenarios and the effects of faulty components [78, 114]; it is used in conjunction with dependability forecasting techniques, in order to populate dependability models with measures obtained from experiments [54, 64, 90]; and to benchmark alternative systems or design choices [62]. The realistic emulation of software faults is a key objective to achieve accurate dependability measures and to investigate faulty scenarios that the

[1] In this work, we follow the notion that a software fault is a development fault originated during the coding phase [13, 36].

system could face during operation. One of the most popular SFI technique is G-SWFIT (Generic Software Fault Injection Technique), proposed by Durães and Madeira [36]. G-SWFIT injects software faults by mutating the binary executable code of a program. This technique is attractive for practitioners, since it allows to perform Software Fault Injection when the source code is not available, which is often the case when third-party software is adopted. G-SWFIT defines which types of software defects have to be introduced in order to realistically emulate a faulty software, based on recent field data studies that characterized residual software faults in complex systems [25, 36, 105].

An important issue concerning the injection of software faults at binary level is the *accuracy* of the injection campaign, that is, the degree of confidence that a fault injected in the binary code correctly emulates a software defect in the source code. For instance, if we aim to emulate the absence of a variable assignment in the source code, we could remove a "move" instruction at binary level. But, if we consider the emulation of a bug in a C preprocessor macro (i.e., a piece of source code that is replicated several times in the binary code), the problem cannot be resolved by simply looking at the binary code. Therefore, it is important to assess the accuracy of binary-level SFI in order to be effectively adopted in real-world scenarios. Unfortunately, only a few studies evaluated the accuracy of binary-level SFI, which were limited to small programs or to a small number of faults [34, 36, 60], and no previous work analyzed this problem comprehensively.

This chapter provides the state of the art on Software Fault Injection providing more details about fault injection techniques based on code mutation. Then, a method for the experimental evaluation of the accuracy of binary code mutation tool is presented. Then, based on the obtained results, it is described a method developed for the automatic detection of the inaccuracy at binary level.

3.2 Background and Related Work

In order to emulate software faults in fault injection experiments, a model of software faults that can realistically occur in the system under test is required. This property, which is referred to as *representativeness*, is desirable when dependability measures have to be quantitatively assessed, such as coverage factors of fault-tolerant systems [54, 64], which depend on the probability distribution of faults and workloads [90]. Fault representativeness is also important to stimulate the complex failure modes that can be exhibited by a software system or component, which are potentially more subtle than simple process hangs or crashes and are not necessarily known a priori [85, 114]. Field data studies analyzed software faults in complex software systems, and can be used to define software fault models. Sullivan and Chillarege [105] analyzed a large set of software-related failure reports collected from the MVS OS, and proposed a classification scheme for software faults, which are described in a level of detail close to the programming level. That work was later extended in [24] where the Orthogonal Defect Classification (ODC) and the notion of defect type are introduced.

This notion points to a high-level classification of faults including Function, Checking, Assignment, Algorithm and Interface faults. ODC was aimed at providing feedback during development; the work presented in [36] extends this level of description and proposes a classification scheme that was precise enough for automated fault emulation (e.g., for the "assignment" class of faults, it specifies if the assignment is an initialization, and if an expression or constant is involved). It also presents a field data study where it is pointed out that most of the software faults found in the field belong to the set of fault types shown in Table 3.1, and that they tend to follow a generic fault distribution.

Table 3.1: Fault Types (see also [36]).

Fault Type	**Description**
MFC	Missing function call
MVIV	Missing variable initialization using a value
MVAV	Missing variable assignment using a value
MVAE	Missing variable assignment with an expression
MIA	Missing IF construct around statements
MIFS	Missing IF construct + statements
MIEB	Missing IF construct + statements + ELSE construct
MLAC	Missing AND in expression used as branch condition
MLOC	Missing OR in expression used as branch condition
MLPA	Missing small and localized part of the algorithm
WVAV	Wrong value assigned to variable
WPFV	Wrong variable used in parameter of function call
WAEP	Wrong arithmetic expression in function call parameter

Another aspect affecting the effectiveness of Software Fault Injection is represented by the method adopted to introduce software faults into a system. In fact, SFI requires more complex modifications of the program code/state than simply a bit-flip/stuck-at: the comparison between real software faults and faults injected by SWIFI tools [58, 72] revealed that hardware fault models cannot accurately emulate software faults. The emulation of software faults requires that what it is injected reproduces the intended fault model, i.e.the *accuracy*), in order to correctly evaluate the effects of software faults on the system. Several methods have been devised for emulating software faults, most of them based on rather indirect approaches (i.e., emulating the possible effects of software faults instead of injecting actual faults in the software code).

Past work on software fault injection can be divided in three categories, according to what is actually injected: data errors, interface errors, and code changes (summarized in Table 3.2).

Data errors. This approach consists of injecting errors in the data of the target program (i.e., a deviation from the correct system state [13]). This is an indirect form of fault injection, as what is being injected is not the fault itself but only a possible effect of the fault. The representativeness of this type of injection is difficult to assert, as the relationship between data corruption and its possible root-cause (i.e., faults) is difficult to establish. However, data errors are an useful and practical means for inducing software failures and debugging of fault-tolerance mechanisms [114].

Interface errors. This approach is in fact another form of error injection where the error is specifically injected at the interface between modules (e.g., system components, or functional units within a program). This usually translates to parameter corruption in functions and API, and it is considered a form of robustness testing. The errors injected can take many forms: from simple data corruption to syntactically valid but semantically incorrect information. As with data errors, the representativeness of the errors injected at the interfaces is not clear and there is some empirical evidence that supports the idea that injecting interface errors and changing the target code produces different effects in the target [77]. This approach is complementary to the injection of actual software faults, and it has proven to be useful to find interface weaknesses [67].

Code changes. Changing the code of the target component to introduce a fault is naturally the closest thing to having the fault there in the first place. However, this is not easily achieved as it requires to know exactly where in the target code one might apply such change, and what instructions should be placed in the target code. Several works followed this notion, although with some limitations: Ng and Chen [85] and the FINE [64] and DEFINE [63] tools use code changes (e.g., changing the destination address of an assignment), although their fault model is very simple and its representativeness is not assured. Madeira et al. [72] showed that SWIFI can be used to inject simple code changes in running processes but cannot emulate more complex software faults. The G-SWFIT technique [36] was developed to address software fault representativeness, by injecting software faults according to the set of most common fault types (Table 3.1) observed in field data.

Table 3.2: Classification of Fault Injection Tools

Category	**Tools**
Data errors	FIAT [15], FERRARI [61], PSN [114], csXception [21], NFTAPE [103], GOOFI [8]
Interface errors	BALLISTA [67], RIDDLE [47], MAFALDA [11], Jaca [74], csXception [62]
Code changes	Ng and Chen [85], FINE [64], DEFINE [63], G-SWFIT [36]

3.2.1 G-SWFIT

G-SWFIT injects code changes at the executable (binary) level (Figure 3.1a). It consists of a set of *fault operators* that define *code patterns* (i.e., a sequence of opcodes) in which faults can be injected (e.g., an MIA fault can be injected wherever an IF construct is found), and *code changes* to be introduced (e.g., the removal of instructions related to an IF construct) to emulate software faults[2]. The proposed fault operators inject valid faults in terms of programming language (i.e., mutated code is syntactically correct) and provide a set of constraints to exclude fault locations that are not realistic (e.g., to inject an MIA fault, the IF construct must not be associated to an ELSE construct, and it must not include more than five statements or loops). The description of a fault operator is provided in Table 3.3. As discussed in the rest of this paper, it is not trivial to assure the accuracy of software

[2] Each fault operator is related to a specific fault type and is denoted with the "O" prefix (e.g., the OMIA fault operator is related to the MIA fault type).

fault injection at the binary level, due to the gap between software faults at source code level (e.g., defects in a program) and their conversion to binary level (i.e., translation of the faulty code in machine code). The implementation of G-SWFIT and the definition of fault operators are dependent on the hardware architecture, the compiler of the target application, and compiler optimizations, since the binary translation of a programming construct (e.g., an IF construct) varies with the compiler and the hardware platform in which the software can be executed. G-SWFIT was originally implemented and applied on the i386 hardware architecture and the Microsoft Windows environment [35]. The technique has then been ported to inject faults in the bytecode of Java programs [96]. Analysing G-SWFIT, it was considered C language with respect to the PowerPC hardware architecture and the GCC compiler, which has been implemented in a R&D tool by Critical Software.

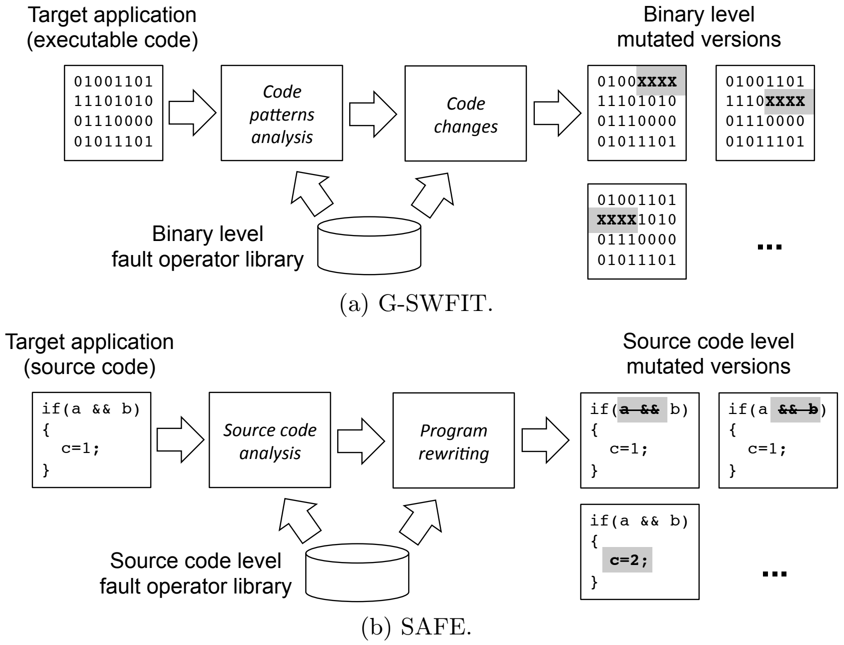
Table 3.3: Description of OMFC Fault Type

Example	*function(...);*
Example with faults	~~*function(...);*~~
Code pattern	*CALL target-address*
Code change	*CALL* instruction removed
Contraints	Return value of the function must not be used (C01) Call must not be the only statement in the block (C02)

3.2.2 SAFE

An alternative approach to change the code of a program consists in mutating its source code, and then to compile the faulty source code to obtain a faulty version (Figure 3.1b). This approach has been implemented in a fault injection tool developed by MOBILAB research group [4], namely SAFE (SoftwAre Fault Emulation) tool. The tool adopts the same fault types of G-SWFIT (Table 3.1), including code patterns and constraints, although faults are introduced in the source code instead of the binary code. This tool has different objectives than G-SWFIT, since it cannot perform fault injection when the source code is not available; it is considered as a support to evaluate the accuracy of G-SWFIT. In order to use the SAFE tool, a C preprocessor translates C macros in a source code file (e.g., inclusion of header files) to produce a self-contained compilation unit. A C/C++ front-end then processes the compilation unit, in order to produce an internal representation of the program (Abstract Syntax Tree, AST). The tool searches for suitable fault locations in the AST and applies a fault operator if all constraints are met, e.g., to inject a MIFS fault, an IF construct should not contain more than 5 statements. The tool produces a set of faulty source code files, each containing a different software fault. The faulty version is obtained by replacing a source code file with a faulty file and recompiling the program.

Compared to the binary level approach followed by the original G-SWFIT, the source code level approach assures the accurate emulation of fault types, since full information about programming constructs and variables is available (this information is missing and has to be reconstructed when injecting faults at the binary level). Moreover, injection in the source code is portable among all platforms in which the target program can be compiled,



(a) G-SWFIT.

(b) SAFE.

Figure 3.1: Software Fault Injection Techniques

without any additional efforts to adapt the fault injection tool to different hardware or compilers. The drawbacks of this approach are that it increases experiment time, since the program needs to be compiled after the injection of a fault, and that the approach cannot be adopted when the source code is missing.

3.3 Experimental Evaluation of Binary Fault Injection

The experimental evaluation is motivated by the fact that the accuracy of binary-level fault injection is limited by the impossibility to correctly recognize some programming constructs in a binary program. The evaluation of binary-level fault injection in a real-world system contributes to understand the limitations and the accuracy of the results that can be obtained by a fault injection campaign.

An example of a wrongly injected fault is represented by a C program containing a SWITCH construct with two branches; in some architectures and compilers (this is the case of GNU GCC compiler for PowerPC architectures), the SWITCH may be translated in binary code using the same opcode sequence of an IF-ELSE construct, since they both consist of a logical condition (which is translated using an opcode that compares two values) and two branches (which are translated using branch opcodes). Therefore, a MIEB (see Table 3.1) fault could erroneously be injected in a code location in which there is not an IF-ELSE construct. It may also happen that a code location suitable for fault injection cannot be recognized in the binary code. For instance, a compiler may translate a function call as inline code (i.e., the function call is replaced with the body of the called function); in this case, a fault injection tool would not be able to recognize the function call, thus omitting to inject an MFC fault in that location. The experimental validation aims to assess the relative occurrence of this kind of problems in real-world complex software, in order to evaluate whether G-SWFIT can achieve an acceptable degree of accuracy even in the presence of these problems. Although some of these problems are already known, their extent in large and complex software has not been investigated in previous studies.

The method also aims at pointing out issues that may arise when implementing G-SWFIT, by highlighting cases in which faults are not correctly injected. Binary-level fault injection tools are difficult to implement, since they have to encompass all potential ways in which programming constructs are translated. This problem is further exacerbated if it is considered the complexity of modern CPUs, programming languages and compilers (whose inner working is usually unknown). Thus it is likely that developers may neglect some code patterns, thus leading to design errors in the fault injection tool.

The proposed method evaluates the accuracy of G-SWFIT by comparing the faults it generates with the ones injected in the source code. Indeed, since a software fault is a defect in the code of a program, it is clear that fault injection at source code level is more accurate. Based on this consideration, faults injected by the two techniques are compared and classified faults in the following three categories:

1. *Correctly Injected faults*: correct faults generated by both techniques. The larger is the set of common faults, the higher is the accuracy of G-SWFIT.

2. *Omitted faults*: faults injected only at source-code level. They correspond to programming constructs in which a fault could exist, but which have not been identified in the binary code.

3. *Spurious faults*: faults injected only by G-SWFIT at binary level that do not match any fault at source-code level. Therefore, they are not considered as representative software faults.

It is important to note that source-level faults can be used as a term of comparison for binary-level faults because (i) *the same fault types are adopted for both binary- and source-level fault injection* (shown in Table 3.1), and (ii) *binary- and source-level faults are injected in every potential location* (i.e., fault injection campaigns are exhaustive). The method (depicted in Figure 3.2) consists of two phases, namely (i) automatic matching of binary-level and source-level faults (Section 3.3.1), in order to identify Correctly Injected faults, and (ii) fault sampling and manual analysis (Section 3.3.2), in order to identify which issues affect the accuracy of G-SWFIT. As a real-world case study, it is considered CDMS (Command and Data Management System), a real-time embedded system developed by Critical Software for the space domain (Section 3.3.3).

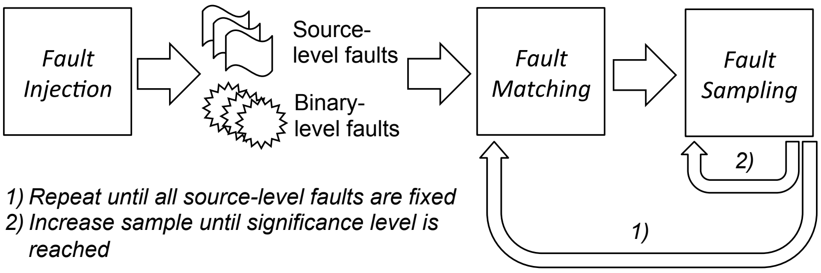
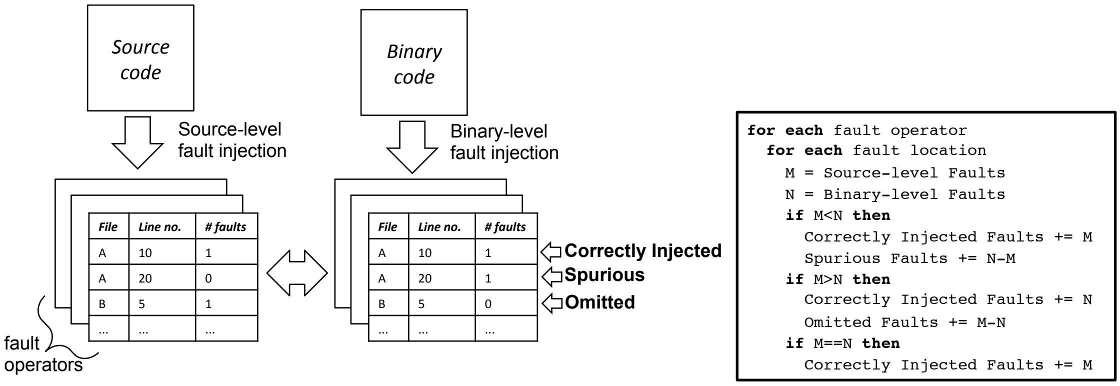

Fault Injection
Source-level faults
Binary-level faults
Fault Matching
Fault Sampling
2)
1)
1) Repeat until all source-level faults are fixed
2) Increase sample until significance level is reached

Figure 3.2: Overview of the Method for G-SWFIT Evaluation

3.3.1 Fault Matching

Fault Matching is based on the assumption that if both techniques inject the *same fault type* in the *same location* (e.g., an assignment or function call is removed both in the source code and in the corresponding location in the machine code), then they are injecting the same fault. It is reasonable to make this assumption since if a fault location is identified both at the binary and source levels, then that fault location is valid and correctly handled. In order to be sure that this assumption holds (and therefore the results are valid), a sample of Correctly Injected faults using the Fault Sampling procedure (explained in the next subsection) manually analyzed. Following this observation, binary-level and source-level faults are compared with respect to their fault types and their locations in the source code (i.e., the source file, the function and the line of code in which a fault is injected). A binary-level fault matches a source-level fault *if they have the same fault type and they are injected in the same code location* (compared using debug symbols in binary code).

The procedure shown in Figure 3.3 has been adopted to identify Correctly Injected faults. If a binary-level fault matches a source-level fault, and only one binary-level fault and only source-level fault exist for the code location under analysis, then the binary-level fault is considered as Correctly Injected. In some cases (e.g., when there are more than one statement in the same line of code), more than one binary-level fault (N), or more than one source-level fault (M) may occur in the same code location. If there are more binary-level faults than source-level faults in the same location ($N > M$), then there are M Correctly Injected faults, and $N - M$ Spurious faults. Similarly, if source-level faults are more than binary level faults ($M > N$), then there are $M - N$ Omitted faults. It follows that if a binary-level fault does not match any source-level fault, then it is considered a Spurious fault, and that if a source-level fault does not match any binary-level fault, then it is considered an Omitted fault. In the examples of Figure 3.3, the proposed procedure identifies one Correctly Injected fault (location *A-10*), one Spurious fault (location *A-20*), and one Omitted fault (location *B-5*).



File	Line no.	# faults
A	10	1
A	20	0
B	5	1
...	...	...

File	Line no.	# faults
A	10	1
A	20	1
B	5	0
...	...	...

```
for each fault operator
  for each fault location
    M = Source-level Faults
    N = Binary-level Faults
    if M<N then
      Correctly Injected Faults += M
      Spurious Faults += N-M
    if M>N then
      Correctly Injected Faults += N
      Omitted Faults += M-N
    if M==N then
      Correctly Injected Faults += M
```

Figure 3.3: Fault Matching Procedure

3.3.2 Fault Sampling

After the Fault Matching procedure, it is performed a detailed analysis of faults in order to investigate the causes of Spurious and Omitted faults, and to verify that Correctly Injected faults are actually correct. Moreover, the aim is to understand whether Omitted and Spurious faults are due to inherent limitations of G-SWFIT or not. Indeed, these faults may occur due to design issues in G-SWFIT as previously discussed; the identification of these issues is useful to provide guidelines for improving G-SWFIT, and to obtain a more precise figure of merit of the G-SWFIT technique. For these reasons, a random sample of Omitted and Spurious faults is analyzed and classified into the following categories:

1. *C preprocessor macros.* When the G-SWFIT technique was proposed, preprocessor macros have been recognized as a frequent cause of Omitted and Spurious faults [36]. A preprocessor macro consists of a piece of code that is replicated for each time the macro is referred within the program. Therefore, when a preprocessor macro has a software fault, the faulty code is replicated several times in the binary code. Since the binary code lacks information about macros, G-SWFIT cannot recognize that macro code is replicated elsewhere within the program: therefore, a Spurious fault is injected for each replica of the macro, and source-level faults that could be injected into macro represent Omitted faults since G-SWFIT cannot correctly injected them (see also Figure 3.4).

2. *Inline functions.* In a similar way to preprocessor macros, inline functions are replicated each time the function is called within the program. Since G-SWFIT does not recognize inline functions within binary code, they lead to Spurious and Omitted faults as well.

3. *Various causes.* This category includes all the other causes of Spurious and Omitted faults that are not related to macros or inline functions.

4. *Issues in the SAFE tool.* Even if source-level fault injection can be considered accurate, it is not excluded the possibility that the adopted source-level fault injection tool could inject faults incorrectly. Therefore, during the manual analysis, it is also looked for issues in the SAFE tool that caused faults to erroneously appear as Spurious or Omitted faults. Since it is required to assure that source-level faults are correctly injected, fixes were made in the SAFE tool when an issue is found and repeat the whole analysis (including both Fault Matching and Fault Sampling) until this category becomes empty.

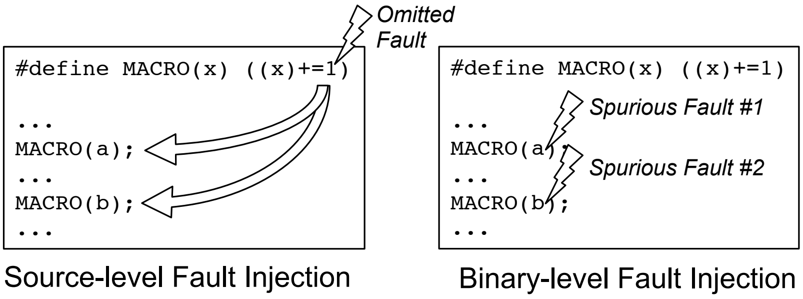

Omitted Fault
#define MACRO(x) ((x)+=1)
...
MACRO(a);
...
MACRO(b);
...
Source-level Fault Injection
#define MACRO(x) ((x)+=1)
Spurious Fault #1
...
MACRO(a);
Spurious Fault #2
...
MACRO(b);
...
Binary-level Fault Injection

Figure 3.4: Examples of Spurious and Omitted Faults Due to C Preprocessor Macro

Because of the high number of the generated faults, the manual analysis is conducted on a sample of faults and then conclusions are drawn about the whole set of faults. In order to generalize the results from the sample, it was addressed the problem of choosing a sample of appropriate size, such that it could be considered representative of a population with more than two categories (i.e., a multinomial distribution, where it is defined π_i as the proportion of the ith category). The sample should be large enough to assure that all of the estimated proportions π_i are within a given confidence interval with significance level $1-\alpha$.

Assuming that the population and the sample are large enough to use the normal approximation, the probability α_i that the proportion π_i lies outside an interval of width $2d_i$ is given by (see [110] for more details about sampling)

$$\alpha_i = Pr\left\{|Z_i| \geq d_i\sqrt{n}/\sqrt{\pi_i(1-\pi_i)}\right\} \tag{3.1}$$

where $1 \leq i \leq k$ and Z_i is a standard normal random variable. By Bonferroni's inequality [110], the probability that one or more of the k estimates will fall outside its interval will be less than or equal to $\sum_i^k \alpha_i$. Equation (3.1) allows to assess if the sample size is large enough to achieve accurate results. If $\sum_i^k \alpha_i > \alpha$, then a larger sample size is required, otherwise the estimated proportions are considered accurate.

This method was applied to the populations of Omitted and Spurious faults by considering $k = 4$ categories (C preprocessor macros, inline functions, various causes, issues in the SAFE tool), assuming a confidence interval of half-width $d_i = 0.05$ and a significance level $1-\alpha = 0.9$. This method was also applied to the population of Correctly Injected faults, in order to analyze whether they are truly correct or not ($k = 2$ categories are considered). For each population, a sample of 5% of faults it is extract and manually analyzed in order to obtain an initial estimate of the proportions; the sample size is gradually increased and analyzed until the required significance level is reached.

3.3.3 Case Study

The case study is a satellite data handling system named Command and Data Management System (CDMS). A satellite data handling system is responsible for managing all data transactions (both scientific and satellite control) between ground system and a spacecraft (Figure 3.5), based on the ECSS-E-70-41A standard [39] adopted by the European Space Agency. In this system, a space telescope is being controlled and the data collected is sent to a ground system. As shown in the Figure, the CDMS, which executes on the spacecraft (*on-board system*), is composed by several subsystems: the TC Manager receives a series of commands from the ground control requesting telemetry information; the TM Manager sends back telemetry information for each command sent; the other modules (PC, PL, OBS, RM, DHS) perform tasks for the data management and the telescope handling. The importance of the *accuracy* of SFI in mission-critical systems like CDMS has been demonstrated in [78], in which two OSs (RTLinux and RTEMS) were compared with respect to the risk of failures of the CDMS due to OS faults, in order to select the most reliable OS for this scenario.

The CDMS application was developed in C and runs on top of an open-source, real-time operating system, namely RTEMS[3]. The CDMS makes use of the RTEMS API for task management, communication and synchronization, and for time management. This software

[3] `http://www.rtems.org`

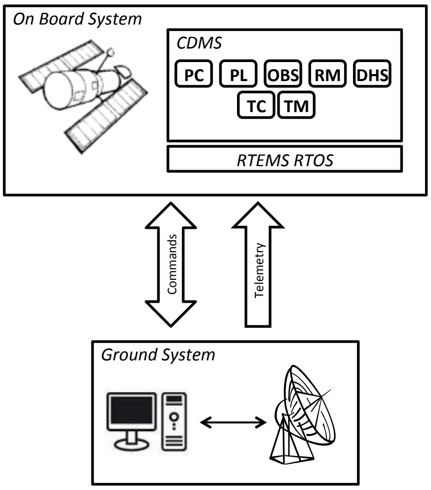


Figure 3.5: Architecture of the Case Study

system is compiled to run on a PowerPC hardware board by using the GCC compiler and disabling compiler optimization settings, which is the setup currently supported by the G-SWFIT tool.

The analysis is focus on faults injected in both the OS (i.e., RTEMS) and application (i.e., CDMS) code. *We only consider the code which is actually compiled and linked in the executable running on the on-board system.* A small part of the code (1.90%), which is written in assembly language to provide board-specific support, is not targeted by our source-level fault injection tool, but its influence on the results can be considered negligible.

3.3.4 Results

In this section, software faults injected at the binary and source-level in a complex case study are analyzed using the method proposed in Section 3.3. Faults at the binary level were generated with the G-SWFIT technique, by using a R&D prototype tool provided by Critical Software company. Faults at the source code level were generated using the SAFE fault injection tool (described in Section 3.2.1). In total, 18,183 source-level faults and 12,380 binary-level faults were generated, respectively. Their distribution across fault types is shown in Figure 3.6. The two distributions exhibit noticeable differences: more source-level faults are injected with respect to some fault operators (such as OMLPA, OWVAV, OWPFV, and OWAEP), whereas in other cases more binary-level faults are injected (such as OMIEB and OMVA, where the latter groups together the OMVAV, OMVIV, and OMVAE operators).

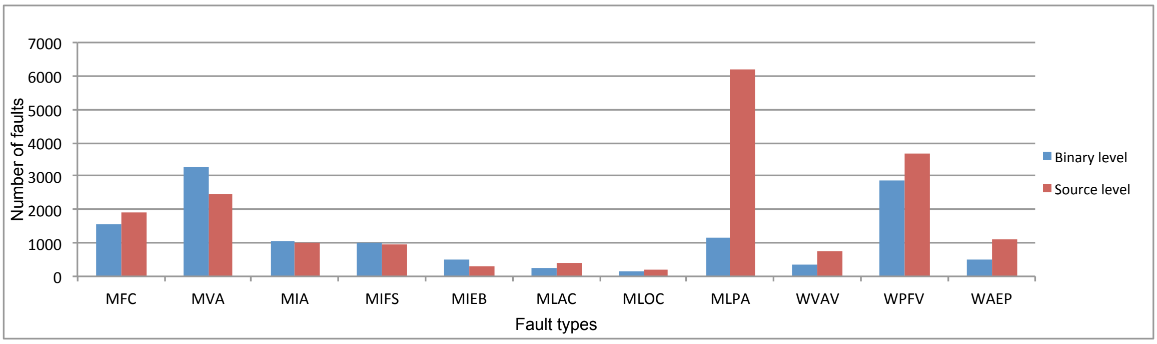


Figure 3.6: Distributions of Software Faults at both Binary and Source Code Level

The Fault Matching procedure (Section 3.3.1) identified the subset of Correctly Injected faults (i.e., common to both techniques) that are analyzed in order to assure the correctness of the method. Correctly Injected faults have been sampled (see Section 3.3.2), and then compared by looking at i) the faulty binary-code generated by G-SWFIT, and ii) the one produced by faults injected in the corresponding source-code locations. This analysis revealed that the binary-level faults match the source-level faults for each fault types and for each sampled faults, except the OWPFV operator. It is found that 40.69% of OWPFV faults at the binary level do not match OWPFV faults at the source-code level even if they affect the same locations, since there are several functions parameters and possible replacements for a given location. In order to take into account this aspect, results shown in Figure 3.7 have been updated by reducing the number of Correctly Injected faults for the OWPFV operator and increasing the number of Omitted and Spurious faults by the same amount.

Correctly Injected faults turned out to be 5,927 (Figure 3.7). They represent 47.88% of faults injected by G-SWFIT. The remaining faults injected by G-SWFIT (52.12%) in the binary code do not match to a software fault in the source code, therefore most of G-SWFIT faults are Spurious. Correctly injected faults represent 32.60% of faults injected in the source code, so the remaining faults at the source level (67.40%) are not emulated by G-SWFIT and they result as Omitted faults. The experimental campaign confirms that the accurate injection at the binary level is a challenging task, at least when a complex software system is considered.

The distribution of the causes of inaccuracies (for both Omitted and Spurious faults) are presented in Figure 3.8. These distributions have been obtained by applying the sampling procedure described in Section 3.3.2. Most of spurious faults (Figure 3.8b) are caused by C

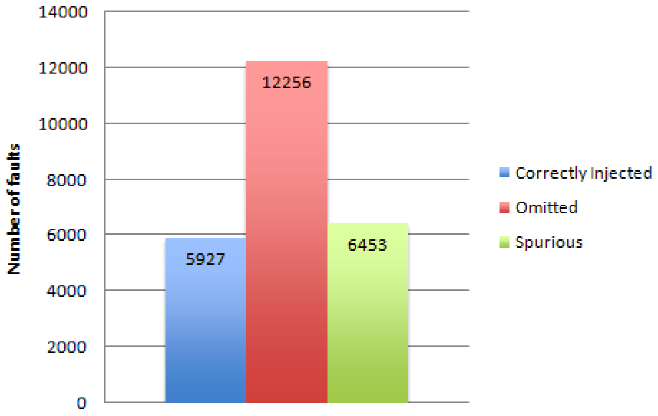
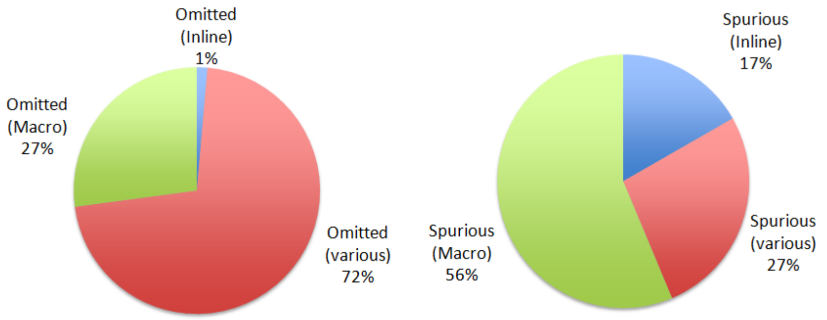


Figure 3.7: Correctly Injected, Spurious, and Omitted Faults

macros (56%) and inline functions (17%). In these cases, every time that a macro or inline function has been replicated in the binary code, G-SWFIT generated an individual binary-level fault; this led to a large number of Spurious faults (i.e., Spurious faults are repeated for each replica of a macro or inline function). In a similar way, macros and inline functions are a noticeable part of Omitted faults (27% and 1%, respectively); this percentage is low when compared to Spurious faults, since one Omitted fault in a macro or inline function leads to several Spurious faults, one for each replica of the code (see also Figure 3.4).

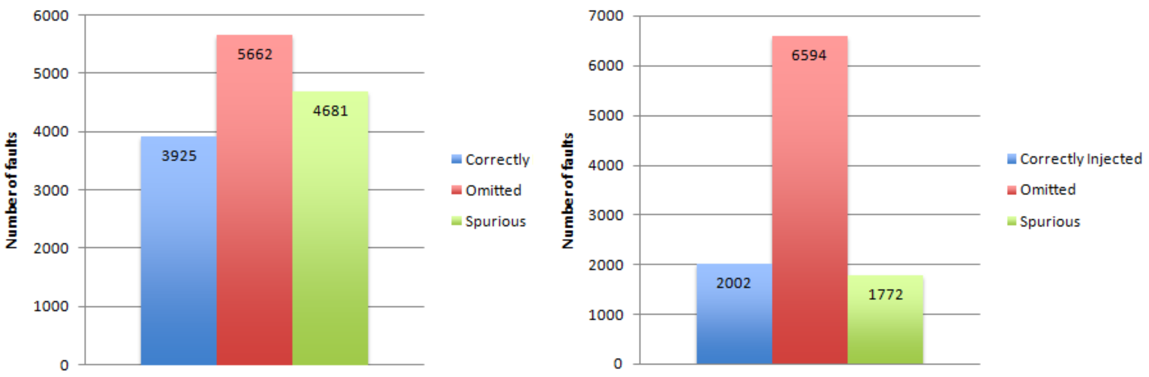
In order to gain more insights into the results, they are separately analyzed the faults injected in the OS and application code, respectively. Figures 3.9 and 3.10 show from a different perspective the data of Figures 3.7 and 3.8, by dividing the results between faults in RTEMS (i.e., OS code) and in CDMS (i.e., application code). It can be noted that faults follow a similar trend in OS and application code, since in both cases the number of spurious



(a) Omitted Faults (b) Spurious Faults

Figure 3.8: Causes of Incorrect Fault Injection in the Case Study

faults is close to the number of correctly injected faults, and the number of omitted faults is predominant. Nevertheless, omitted faults seem to be much more in the case of CDMS (Figure 3.9b).



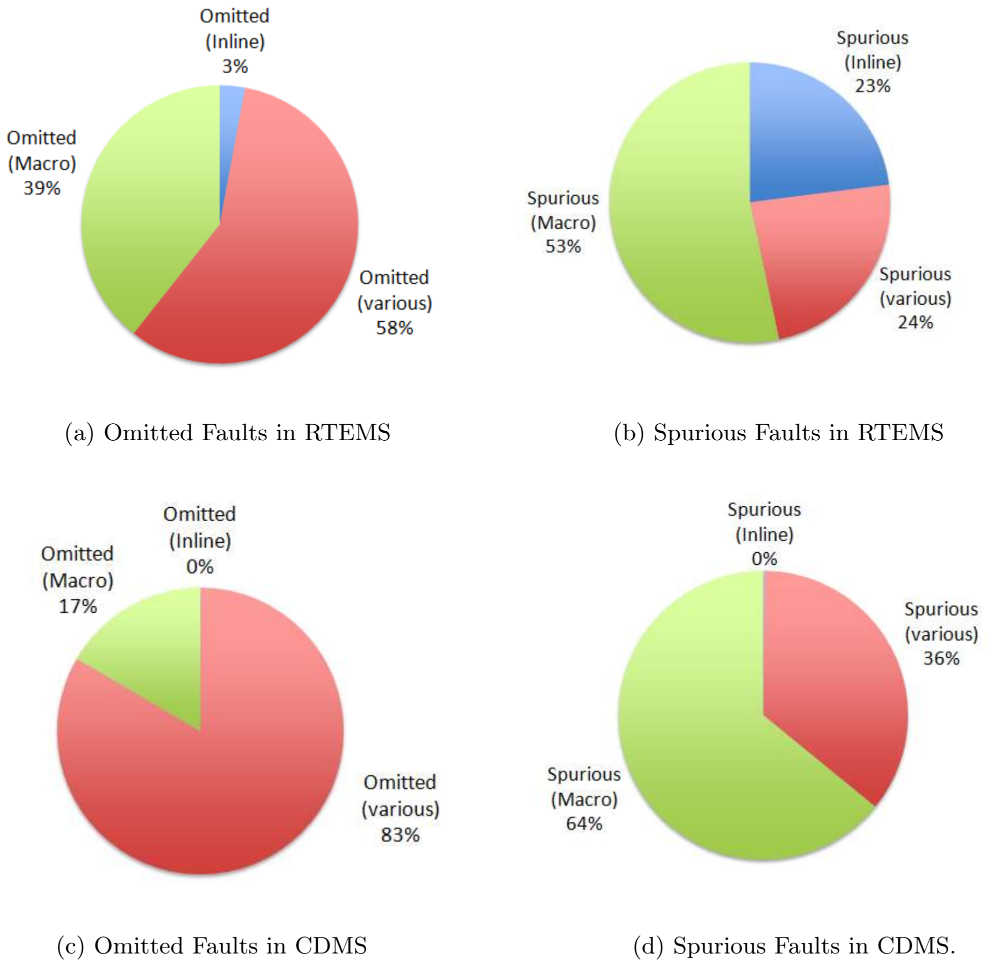
(a) RTEMS Code (b) CDMS Code

Figure 3.9: Number of Faults (Correctly Injected, Spurious, and Omitted) in OS and Application Code

Figure 3.10 shows that omitted and spurious faults due to various causes (i.e., not related to macro or inline functions) are more frequent in CDMS than in RTEMS. The constructs not correctly recognized at the binary level (e.g., see the examples in Figures 3.12 and 3.13 discussed later in this section) likely occur more often in application code due to higher complexity of that code, thus causing an higher number of omitted faults. Moreover, macros and inline functions are more frequent for RTEMS; this is due to the fact that several RTEMS functions are exported as macros and inline functions in order to be used by external code (i.e., user and library code that is compiled and linked with RTEMS code).

Software complexity metrics collected from the case study code (see Table 3.4) confirm that functions in the application code tend to be more complex than those in the OS code (in term of size, cyclomatic complexity and input/output dependencies). This is a common trend in embedded systems, in which the OS is kept as simple as possible in order to reduce the overhead and the number of potential defects [92]. Moreover, the number of preprocessor statements per function confirms that RTEMS makes a more extensive use of macros that CDMS. Therefore, it is even more important to fix the implementation issues mentioned above if a fault injection tool is intended to be used with complex software.

The "various causes" behind spurious and omitted faults are numerous and specific to each fault operator. It is not possible to provide a precise estimate of the relative percentage of each cause, since it would require to manually analyze an extremely large sample of injected faults. Instead, it was identified which part of incorrectly injected faults are due



(a) Omitted Faults in RTEMS

(b) Spurious Faults in RTEMS

(c) Omitted Faults in CDMS

(d) Spurious Faults in CDMS.

Figure 3.10: Causes of Incorrect Fault Injection in OS and Application code

Table 3.4: Comparison of Average Software Complexity Metrics of Functions in RTEMS and CDMS Code

Metric	**RTEMS**	**CDMS**
Lines of Code	17.30	30.71
Preprocessor Statements	0.64	0.15
Cyclomatic number	5.63	6.61
Number of inputs	5.50	7.38
Number of outputs	4.12	6.84

to unavoidable limitations of G-SWFIT, and which of them can be avoided by improving the G-SWFIT fault injection tool. To do so, it was excluded from the sample those faults not related to macros or inline functions, and reason were diagnosed (with the support of Critical Software developers) to understand why omitted faults were not injected, and why spurious faults were erroneously injected. It was found that 26.02% of omitted and spurious faults were due to causes that are impossible to avoid when injecting at the binary code level, including:

- *Low-level translation of C operators.* Some C expressions (like *sizeof* and array and struct accesses using -> and *[]*) are translated by introducing arithmetic operations and constants in the binary code; these operations are recognized as arithmetic expressions by fault operators such as OMVA, OWVAV, and OWAEP.

- *Switch and goto constructs.* These constructs are translated in a similar way to IF constructs using *branches* in the binary code; therefore, IF constructs are not always correctly identified by operators such as OMIA, OMIEB, and OMIFS.

- *Forced function inlining.* Some functions (e.g., *memcpy*, *memset*) are compiled as inline functions, although they are not declared as inline.

Since the binary code lacks information about high-level constructs, the causes mentioned above cannot be avoided. In practice, these inaccuracies have to be accepted as limitations of fault injection at binary level, and should be taken into account when conclusions are drawn from fault injection experiments.

Nevertheless, during the manual analysis it is observed that several Omitted and Spurious faults not related to intrinsic limitations of fault injection at binary level, but were due to limitations of the fault injection tool; they represent the 73.98% of the analyzed sample. These inaccuracies occurred since some checks have not been implemented yet in the tool, and some fault operators diverge in some cases from the fault types encompassed by G-SWFIT due to choices that simplify the implementation. Therefore, part of the Omitted and Spurious faults could be avoided by improving the implementation of binary-level fault injection.

An example of Spurious fault is provided in Figure 3.11, which shows a fault location in the source code (monospace font) along with its machine code translation (italic font). It is a spurious MFC fault in CDMS that has been injected in a wrong location. In this example, the function call should not be removed since it is the only statement within a block of code, and a fault in that location would not be realistic. The OMFC operator imposes a constraint (Table 3.3) to avoid fault injection in this kind of location [36]. Instead, the fault has been injected by the tool since the block containing the function call is not recognized (i.e., the constraint is not enforced by the tool).

Figure 3.12 and Figure 3.13 provide two examples of Omitted faults that were caused

```
static void HousekeepingAction(TmPacket *STm) {
          stwu r1,-24(r1)
          mflr r0
          stw r31,20(r1)
          stw r0,28(r1)
          mr r31,r1
          stw r3,8(r31)

    SendTmMsg(pbtBuffer,
        TmGetPacketTotalLength(STm)); ← MFC fault location
          lwz r3,8(r31)                 (to be avoided)
          bl 00006184 <TmGetPacketTotalLength>
          mr r0,r3
          lis r9,7
          addi r3,r9,-21944
          mr r4,r0
          bl 0000a3b4 <SendTmMsg>

}
          lwz r11,0(r1)
          lwz r0,4(r11)
          mtlr r0
          lwz r31,-4(r11)
          mr r1,r11
          blr
```

Figure 3.11: Spurious MFC Fault in CDMS.

by limitations in G-SWFIT implementation. In Figure 3.12, a function call which could be removed by the OMFC fault operator is not identified. As confirmed by Critical Software developers, the *TcMakePacket* function is not recognized as returning a value that is stored and used later in the program. Therefore, a fault is not injected due to a constraint of the OMFC operator requiring that the return value of a function should not be in use (Table 3.3).

In Figure 3.13, the fault location has been omitted for an even more subtle reason. In this example, the *return* statement within the IF construct is translated with a branch to the end of function, and the tool incorrectly believes that the IF construct includes all the statements until the end of the current function. A fault is not injected since the IF construct

```
TcMakePacket(pbtBuffer, &STc);    ← MFC fault location
      addi r0,r31,24                  (not identified)
      lis r9,9
      addi r3,r9,-21492
      mr r4,r0
      bl 000056b8 <TcMakePacket>

bOk = CheckAppIdTypeSubtype(&STc);
      addi r0,r31,24
      mr r3,r0
      bl 00011a10 <CheckAppIdTypeSubtype>
      mr r0,r3
      stw r0,20(r31)
```

Figure 3.12: Omitted MFC Fault in CDMS

should not contain more than 5 statements [36]. Although the tool is provided with checks to avoid these mistakes, a check to avoid this specific case was not implemented. This kind of issue seems to be more relevant for Omitted faults than for spurious faults given the high number of omitted faults due to various causes, as depicted in Figures 3.7 and 3.8.

Other incorrect behaviors were also found in the prototype tool, which were due to the incomplete implementation of constraints or the identification of code blocks and control structures. In Figure 3.14, it is provided an evaluation of the results that can be obtained by improving the mentioned aspects. The improvements prevent the occurrence of several Omitted and Spurious faults: Correctly Injected faults represent the majority of faults potentially injectable in the source code (i.e., only a minor part of faults is omitted), and they also represent the majority of faults actually injected by G-SWFIT (i.e., only a minor part of faults is spurious). It is possible to conclude that the evaluation of a binary-level fault injection tool on real-world complex software is useful to identify implementation issues, and

```
rtems_status_code sc;
n32Size = TcGetAppData(STc, &pbtData);
    lwz r3,120(r31)
    lis r9,7
    addi r4,r9,-23004
    bl 00005934 <TcGetAppData>
    mr r0,r3
    lis r9,7
    stw r0,-22992(r9)

sc = rtems_semaphore_obtain(rtems_mon_Mutex,
                            RTEMS_WAIT,
                            RTEMS_NO_TIMEOUT);
    lis r9,7
    lwz r0,-22948(r9)
    mr r3,r0
    li r4,0
    li r5,0
    bl 0003d504 <rtems_semaphore_obtain>
    mr r0,r3
    stw r0,64(r31)

if (sc != RTEMS_SUCCESSFUL)     ← MIA fault location
    lwz r0,64(r31)                (not identified)
    cmpwi cr7,r0,0
    bne- cr7,0000c69c <AddMonitoringAction+0x97c>
    return;

if ( n32Size >= 10 ) {
    lis r9,7
    lwz r0,-22992(r9)
    cmplwi cr7,r0,9
    ble- cr7,0000c680 <AddMonitoringAction+0x960>
```

Figure 3.13: Omitted MIA Fault in CDMS.

should be adopted to assure that a tool does not omit valid fault locations, and that spurious faults are not generated.

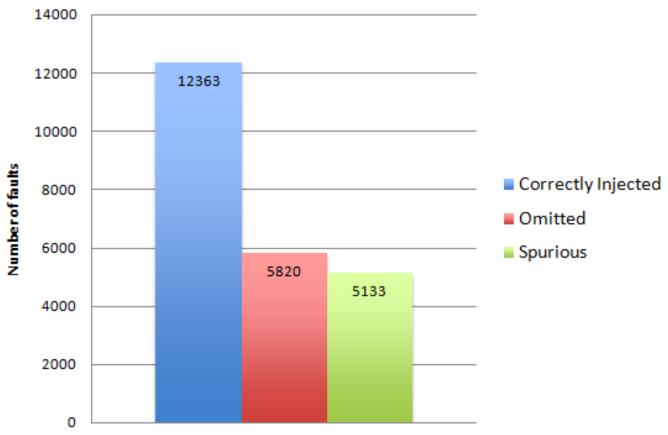


Figure 3.14: Number of Faults (Correctly Injected, Spurious, and Omitted) when Fixing Implementation Issues of the G-SWFIT Tool

3.4 Systematic Testing of Binary Fault Injection

The experimental analysis proved that the implementation of G-SWFIT is tricky. Behind issues related to its dependency on the compiler of the target software, compiler optimizations, and the hardware architecture, binary level code mutation is difficult because the translation of programming constructs (such as loops and control flow constructs) varies with these factors in subtle ways. Moreover, the fault types (see Table 3.5) to be injected require the identification of complex high-level programming patterns, namely *constraints*. Constraints (Table 3.6) are rules defined by G-SWFIT in order to better emulate the types

Table 3.5: Fault Types of G-SWFIT [36].

Acronym	**Description**	**Constraints**
MFC	Missing function call	C01, C02
MVIV	Missing variable initialization using a value	C02, C03, C04, C05, C06
MVAV	Missing variable assignment using a value	C02, C03, C04, C06, C07
MVAE	Missing variable assignment with an expression	C02, C03, C04, C06, C07
MIA	Missing IF construct around statements	C08, C09
MIFS	Missing IF construct + statements	C02, C08, C09
MIEB	Missing IF construct + statements + ELSE construct	C08, C09
MLAC	Missing AND in expression used as branch condition	
MLOC	Missing OR in expression used as branch condition	
MLPA	Missing small and localized part of the algorithm	C02, C10
WVAV	Wrong value assigned to variable	C03, C04, C06
WPFV	Wrong variable used in parameter of function call	C03, C11
WAEP	Wrong arithmetic expression in function call parameter	

of fault found in the field [36]. For instance, the MFC fault type has a constraint (C01) imposing that a function call should removed only if it does not return any value or the return value is discarded. In fact, field data (and also intuition) suggest that removing a function call whose return value is used later in the program would not be representative of real software faults made by programmers. Another example is the MLPA fault type, which has a constraint (C10) that imposes to remove between two and five consecutive statements, which should not be control or loop statements. These constraints are important for emulating software faults in a representative way, which is a requirement for a correct evaluation of fault tolerance and for dependability benchmarking [80].

Table 3.6: Constraints of Fault Types in G-SWFIT [36]

Id	Description
C01	Return value of the function must not being used
C02	The construct must not be the only statement in the block
C03	Variable must be local
C04	Must be the first assignment for that variable in the module
C05	Assignment must not be inside a loop
C06	Assignment must not be part of a FOR construct
C07	Must not be the first assignment for that variable in the module
C08	The IF construct must not be associated to an ELSE construct
C09	The block must not include more than five statements and not include loops
C10	Statements are in the same block, do not include more than 5 statements nor loops
C11	There must be at least two variables in this module

The experimental evaluation of the accuracy of G-SWIFIT fault injection tool on a complex software system from the space domain [29]. Faults were injected in both OS and application binary code, and binary mutations were compared with mutations performed on the source code following the same rules of G-SWFIT. The comparison revealed several cases in which G-SWFIT did not correctly mutate the binary code (Figure 3.15). The three types of injections identified are: (i) *correct injections*, that is, binary injections that match a source-code injection; (ii) *omitted injections*, that is, potential injections that were missed at the binary level, and were only performed at the source-code level; and (iii) *spurious injections*, that is, binary injections that do not match any valid source-code injections.

Moreover, omitted and spurious injections were divided in two groups: incorrect injections due to intrinsic limitations of G-SWFIT (and of BCM in general), and incorrect injections due to implementation issues of the BCM tool. Many omitted and spurious injections were caused by limitations of BCM that are impossible or very difficult to avoid: For instance, an *inline* C function, which is replicated at each call site in the program, can mislead a BCM tool to inject several spurious faults, that is, one distinct fault for each replica (instead, a real fault in an inline C function would be present in every call site at the same time). Nevertheless, several omitted and spurious faults were not related to limitations of BCM, but were due to the incomplete or simplified implementation of G-SWFIT. In particular, issues were related to the implementation of fault constraints and to the identification of code blocks and control structures. For instance, spurious faults were in some cases incorrectly injected when the target instruction was the only statement within a block, and some faults were omitted when an IF construct included a *return* statement. These issues are not due to G-SWFIT, and could be avoided by conducting a rigorous evaluation and improvement of the BCM tool. When implementation issues are avoided, then omitted and spurious faults represent the minority of cases.

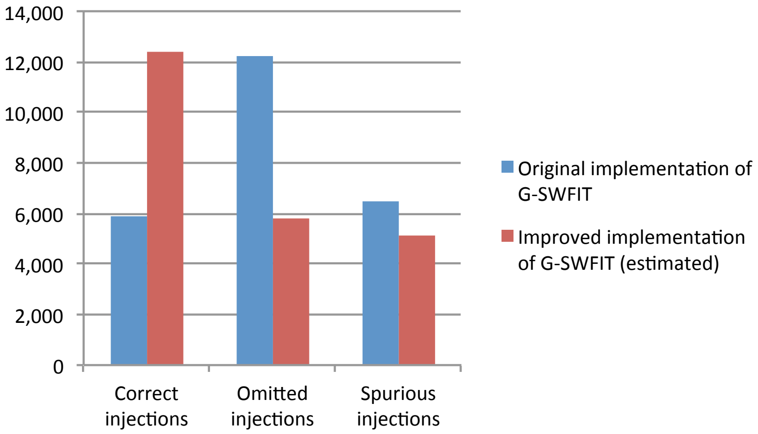


Figure 3.15: Accuracy of G-SWFIT in the Context of an Embedded Space Software [29].

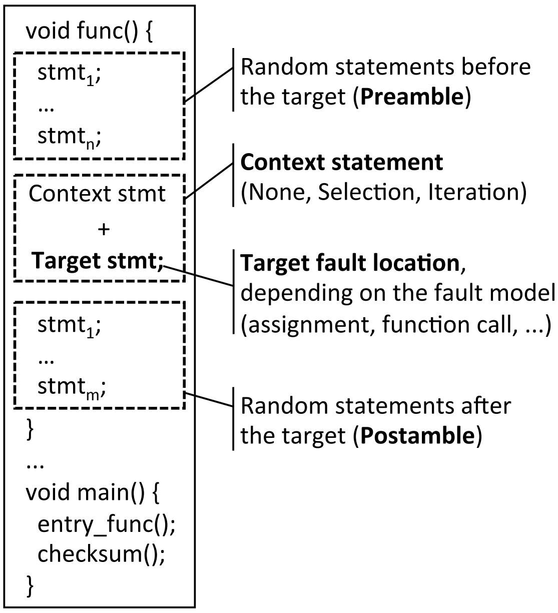
These results motivated the development a systematic approach for testing and improving BCM tools. The experimental methodology (see Section 3.3) [29] cannot be easily adopted by developers of BCM tools, since this methodology is based on the analysis of a large number of injected faults in a large software (tenths of thousands of faults in the experiment), and on the manual analysis of omitted/spurious faults to identify implementation issues. Analyzing even a sample of these faults requires considerable efforts, and provides only a partial evaluation of the BCM tool, since the evaluation would be focused on the code patterns adopted in a specific target program. Therefore, it is proposed an approach that generates, in a controlled and automated way, a (limited) set of small target programs, in order to allow a more efficient evaluation and improvement of BCM tools.

The proposed approach automates the evaluation of BCM tools at injecting faults in binary code. To accomplish this goal, the approach uses *synthetic programs*, that is, programs (in a high-level language, such as C) that are automatically and randomly generated with the sole purpose to evaluate the ability of the BCM tool to inject faults into them. These synthetic programs are then compiled in binary code, and fed to the BCM tool. Finally, mutations obtained from the BCM tool are analyzed.

The key idea of this approach is to control the generation of synthetic program, in such a way to expose the BCM tool to several different code patterns that could point out its limitations. In particular, generated synthetic programs contain a *target fault location* in their code, in which the BCM tool is expected to inject a fault. The target fault location is generated to comply with the fault types and constraints for which the BCM tool is designed. For instance, to evaluate the "missing assignment" fault type of G-SWFIT (see Table 3.1), a target fault location containing an assignment instruction is generated. To comply with the constraints of fault types of G-SWFIT (see Table 3.6), the target fault location consists of an assignment made to a local variable, and this assignment is not the only instruction of its block. If the tool is not able to inject a fault in the target fault location, then the synthetic program exposes an issue of the fault injector, i.e., it exhibits an *omitted injection.*

Moreover, the generation of synthetic programs also encompass programs in which the fault constraints are not fully satisfied, and in which the fault injector should avoid to inject faults. If the fault injector fails to recognize that the target fault location is not compliant to the fault model, then the synthetic program exposes an issue of the BCM tool. This situation represents a *spurious injection*.

To make the synthetic programs more realistic, and to evaluate the accuracy of the BCM tool in the presence of complex code patterns, the target fault location is surrounded, preceded and succeded by additional randomly-generated programming constructs, that represent respectively the *context*, the *preamble* and the *postamble* of the target fault location (Figure 3.16).



```
void func() {
  stmt_1;
  ...
  stmt_n;
  Context stmt
       +
  Target stmt;
  stmt_1;
  ...
  stmt_m;
}
...
void main() {
  entry_func();
  checksum();
}
```

Random statements before the target (**Preamble**)
Context statement (None, Selection, Iteration)
Target fault location, depending on the fault model (assignment, function call, ...)
Random statements after the target (**Postamble**)

Figure 3.16: General Structure of a Synthetic Program

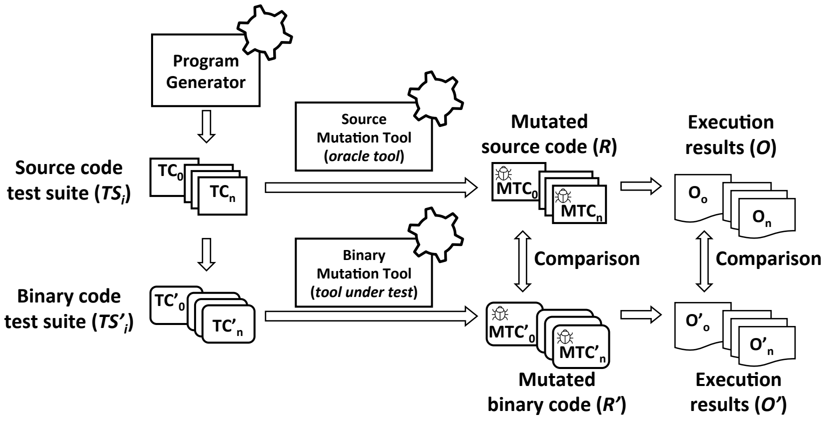


Figure 3.17: Proposed Approach
A program generator generates a test-suite (TS) for a given fault operator. TS' is generated by compiling TS. An oracle tool and the tool under test inject faults in the test-suite, and their results are compared to evaluate the accuracy of the tool under test.

The proposed approach, depicted in Figure 3.17, consists of the following steps:

- **Test-suites generation**: The tool automatically generates synthetic programs. A set of synthetic programs is generated for each fault type encompassed by the tool. These programs are **test-cases** for the BCM tool. A synthetic program is a sequence of randomly generated statements. A statement can be an expression, an assignment, a function call, a selection or an iteration statement. As shown in Figure 3.16, only one statement, named *target*, is considered the fault location, in which faults will be injected. The target statement depends on the fault type and on the constraints that are tested, and it is surrounded by iteration (a loop construct) or selection (a conditional construct) statement, i.e., a *context*. Moreover, the complexity of each synthetic program is controlled by a set of parameters, such as nesting level and type of expressions (e.g., constants, variables, arithmetic operations), as described in Subsection 3.4.1. The set of synthetic programs obtained by varying these parameters make a **test-suite** (*TS*). A test-suite contains both *valid* test-cases (i.e., test-cases satisfying all the constraints for the fault type), and *invalid* test-cases (i.e., test-cases that do not satisfy one of the constraints). Programs in the test-suites are compiled to binary code (*TS'*) before the next phase.

- **Test-suite execution**: The test-suite *TS'* (in binary form) is submitted as input to the BCM tool that are tested (**Tool Under Test**). At the same time, the test-suite *TS* (in source-code form) is fed to another fault injection tool (**Oracle Tool**) that injects faults in source code rather than binary code. The Oracle Tool serves as a term of comparison for evaluating the accuracy of the Tool Under Test. The Oracle Tool adopts the same fault model of the Tool Under Test (e.g., the fault model in Tables 3.1 and 3.6), and it works on source code, thus it avoids the issues encountered by the BCM tool when working on binary code, and is much more easy to implement and accurate than the Tool Under Test.

  The Tool Under Test takes executables in *TS'* as input, and produces faulty versions of these executables (*mutated test cases*, *MTC'*) by changing their binary code. For each test-case in *TS'*, it is collected the faulty executable MTC'_i generated by the Tool Under Test, and the output resulting from the execution of MTC'_i. The same process is performed on the source-code test-suite *TS* using the Oracle Tool.

- **Comparison and detection of inaccuracies**: An analysis is conducted by comparing the mutations that both tools have performed during the injection (*R* and *R'*), and the executions of the mutated test cases (*O* and *O'*). The comparison between *R* and *R'* determines whether the BCM tool did *spurious*, *omitted* or *incorrect* injections. Given a synthetic program, an injection is correct when (i) the Tool Under Test recognizes the target fault location in the synthetic program, and (ii) it mutates the binary code in an equivalent way to the Oracle Tool (denoted by equivalent results from the executions of mutated test cases).

3.4.1 Test-suite Generation

The proposed approach generates synthetic programs using a random program generator tool. It is extended the use of random programs, that were adopted in past studies for testing compilers, to test binary code mutations tools. Program generators adopted in past studies generate programs as a sequence of statements that includes global and local variables declarations, functions, assignment, expressions, selection and iteration statements. The inputs of these programs are constants produced during the random generation process. The output of these programs is a checksum of the global variables of the program, which is computed just before the termination of the program.

For the sake of example, it is implemented a random program generator for the C language, *Faultprog*, which is based on the *Randprog* tool developed by Bryan Turner for testing C compilers [20]. It is extended *Randprog* to support the automatic evaluation of BCM tools. In particular, *Faultprog* produces C programs, called *synthetic programs*, by following rules that depend on the fault model with respect to which the BCM tool is tested.

A *synthetic program* is a sequence of randomly generated statements. A statement can be an expression, an assignment, a function call, a selection or an iteration statement. The random generator bases the program generation on a *stochastic grammar* of the language [75]. A stochastic grammar associates probabilities to each grammar rule describing a language. Each rule consists of a *left side* and a *right side*, where the left side is a non-terminal symbol, and the right side contains one or more sequences of symbols (either terminal and non-terminal). A statement is generated by concatenating *terminal* (e.g., operators like "+" and "->", or keywords like "for") and *non-terminal symbols* in the sequence. Beginning from a "start" rule, the program generator replaces each non-terminal symbol through the recursive application of other rules in the grammar, until there are no more non-terminal symbols. When the right side contains more than one sequence, the stochastic grammar associates a probability to each sequence, and the program generator randomly selects a sequence on the basis of its probability.

In our approach, we control the random program generation to follow the structure showed in Figure 3.16, by appropriately tuning probabilities in the stochastic grammar in the process. The random program should contain a statement, named *target*, that is the location for injecting faults according to the fault model. The elements in the target fault location are selected randomly, according to the following parameters (summarized in Table 3.7):

- *Fault type* that has to be tested. For instance, when testing the MVAE fault type in G-SWFIT, the target fault location consists of a local variable assignment with an expression, such as an arithmetic expression.

- *Fault constraint* to be violated (if any). For instance, when testing the MFC fault type in G-SWFIT, we can generate *valid* programs that comply to both constraints *C01* and *C02*, and *invalid* programs that violate one of these two constraints (e.g., the target statement is function call whose return value is used in the rest of the program).

- *Context* in which the target statement has to be inserted. It can be a *selection* (e.g., if-then-else statement) or an *iteration* statement (e.g., while- or for-loop).

- *Nesting* depth of the context, such as the number of nested loops in which the target statement should be contained.

- *Elementary operand type* (EOT) to use in expressions of the target statement. They can be constants, global or local variables, or function calls.

- *Complexity* of expressions in the target statement. According to this parameter, the target statement contains expressions that are obtained by different combinations of one or more EOTs, random sub-expressions and random operators.

Table 3.7: Parameters of the *Faultprog* random program generator.

Parameter	**Description**	**Options**
Fault type	Type of target statement according to the fault model.	MFC; ... WAEP

Fault constraint	Constraint to be violated (if any) in the target.	all satisfied;
		C01 not satisfied;
		...

Context	The statement surrounding the target statement.	none;
		while;
		for;
		if-target;
		if-target-then-else;
		if-then-else-target

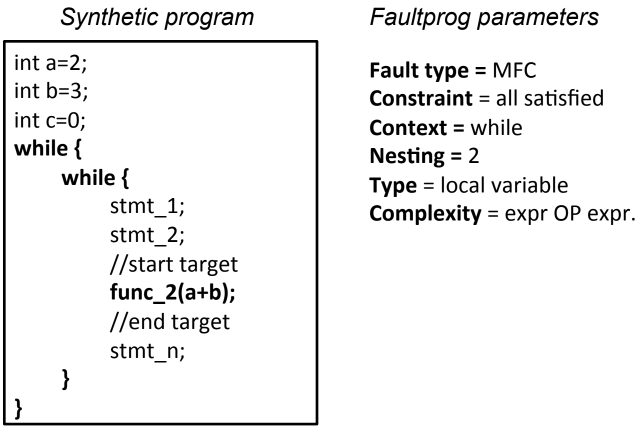
Nesting	The nesting depth of the context statement.	0; 1; 2

Elementary operand type (*EOT*)	The type of operands in expressions of the target.	constant;
		local variable;
		global variable;
		function call

Complexity	Complexity of expressions in the target statement.
	
	NOTE: Expressions are obtained by combining one or more EOTs, random sub-expressions and random operators.
		a simple EOT;
		expr. with two EOTs;
		expr. with three EOTs;
		expr. with an EOTs and a random sub-expr.;
		expr. with an EOTs and two random sub-exprs.

Figure 3.18 shows an example of random (valid) program, in which the MFC fault type

is selected and all constraints are satisfied, a function call is nested in two loops, and the complexity of the expression of the target (in this case, the parameter of the function call) is the sum of two local variables.



Synthetic program

```
int a=2;
int b=3;
int c=0;
while {
      while {
            stmt_1;
            stmt_2;
            //start target
            func_2(a+b);
            //end target
            stmt_n;
      }
}
```

Faultprog parameters

Fault type = MFC
Constraint = all satisfied
Context = while
Nesting = 2
Type = local variable
Complexity = expr OP expr.

Figure 3.18: Example of a Synthetic Program for Testing MFC Fault Type

To obtain test-suites, *Faultprog* automated program generator generates several random programs. Programs are based on different combinations of parameters (Table 3.7), and all combinations of parameters that apply for each specific fault type are considered.

3.4.2 Test-suite Execution, Comparison and Detection of Inaccuracies

Once test-suites are generated (both in source-code form, *TS*, and in binary-code form, *TS'*), it is possible to run them over the Oracle Tool and the Tool Under Test and, for each synthetic program, to store faulty executables produced by the tools, which populate the sets *R* and *R'*.

For each binary executable *TC'$_i$*, the analysis focuses on the binary instructions corresponding to the target fault location of the synthetic program. These instructions can be identified using a-priori knowledge about the generated program, and debugging information that can be introduced into executables by means of the compiler [46]. The executables from both the Oracle Tool and the Tool Under Test are compared to identify *omitted* or *spurious* injections by the Tool Under Test. This matching is based on the observation that both tools should ideally inject the *same fault types* in the *same locations* (e.g., an assignment or function call is removed both in the source code and in the corresponding *move* or *branch* instructions in the binary code).

More precisely, the Tool Under Test behaves correctly (i.e., it passed a test-case) in two cases:

- The generated program is valid, i.e., it contains a target statement that satisfies all the constraints imposed by the fault type being tested. Both tools identify the target statement as a valid location where to inject a fault. The injection is performed by both tools, where *MTC$_i$* and *MTC'$_i$* are equivalent.

- The generated program is invalid, i.e., it contains a target statement that not satisfies one of the constraints imposed by the fault type being tested. Both tools identify the target statement as an invalid location where *not* to inject a fault. The injection is not performed by both tools, so *MTC$_i$* and *MTC'$_i$* are not produced.

On the contrary, a test-case fails when:

- The generated program is valid. The Oracle Tool injects a fault in the target fault location, while the Tool Under Test does not inject a fault in the corresponding binary instructions. The syntactic program detected an *omission* of the Tool Under Test.
- The generated program is invalid. The Oracle Tool *does not* inject a fault, while the Tool Under Test injects a fault in the binary instructions of the target location. The syntactic program detected a *spurious* injection of the Tool Under Test.
- The generated program is valid. Both tools inject a fault, but the result of executions of the two faulty executables are different.

In some scenarios, in which the target fault location is a complex statement, more than one fault could be potentially injected in that location. For instance, this is the case of the "missing arithmetic expression in function parameter", when the target fault location contains several parameters. In that case, the *number* of faulty versions generated by the two tools is compared, and *omitted* or *spurious* injections are identified if the Tool Under Test generated, respectively, less or more injections than the Oracle Tool. This situation highlights the importance of comparing the Tool Under Test with an Oracle Tool at the source-code level: since synthetic programs are generated in a random way, it is needed to compute the number of faults potentially injectable in the target location, by using a static analysis of the code that, in the proposed approach, is offered by the Oracle Tool.

Finally, when both the Tool Under Test and the Oracle Tool inject the same fault type in the target fault location, the two faulty versions are executed (i.e., the one obtained from the Tool Under Test, and the one from the Oracle Tool), and their outputs are compared. Given that the two tools should inject the same fault type, it is expected that they inject faults that have the same effects on the target program, in terms of impact on the control flow of the execution and on the state of the program. Differences are detected in the two executions by computing a *checksum* of all global variables just before their termination, and by comparing these two checksums. Since the synthetic programs make extensive use of global variables, both in the function with the target location, and in other functions preceding or succeeding that function, then analyzing the state of global variables at the end of an execution is likely to reveal discrepancies between the tools. For instance, if the fault affects a local variable that is later used in a control flow condition, then an "incorrectly injected" fault in the binary code would turn in a different control flow than the fault in the source code; in turn, the different control flows would impact on the global variables of the program, thus revealing that the fault in the binary code has been incorrectly injected.

Of course, this approach (or any other approach) cannot *prove* that two faults are equivalent, since this problem is *undecidable*: it is the same problem of detecting *equivalent mutants* in mutation testing [59]. Nevertheless, the focus is different than mutation testing, since each mutant is not going to be executed several times with several test cases, but to be execute only a subset of faults (i.e., only those faults that are neither *omitted* nor *spurious* injections) with few inputs in order to detect the most relevant differences between the Oracle Tool and the Tool Under Test. Thus, the use of checksums is an acceptable and practical solution to this problem.

After performing the comparison of the faulty versions from the Oracle Tool and the Tool Under Test, results are examined to identify the causes of inaccuracies in the Tool Under Test. To give feedback to developers of the Tool Under Test, it is performed an analysis of the *distributions* of failed test-cases with respect to several factors, to identify which factor leads to the highest number of failed test-cases, for instance the fault types, the constraints, or the type of context that causes a significant number of omitted or spurious injections. This information enables developers to pinpoint the causes of inaccuracies, by looking at specific areas of the Tool Under Test. Moreover, after fixing the Tool Under Test, developers can apply again the test-cases in order to validate the fix, i.e., to check whether the fix was able to significantly reduce the number of inaccuracies.

3.4.3 The csXception™ suite

csXception™ is a fault injection tool developed by Critical Software© for supporting the validation activities of safety- and mission-critical systems. It includes several plugins for software-implemented and software fault injection as shown in Figure 3.19. csXception™

was originally designed to perform hardware fault injection by exploiting CPU debugging and performance monitoring features available in modern microprocessors.

More recent is the development of the G-SWFIT based R&D plug-in, i.e. the component that enables injection of software faults based on binary mutations. csXception^{TM} core is the Experiment Management Environment (EME), responsible for controlling, monitoring, and storing results of the experiments. It exchanges data with plug-in that, instead, represents the injector. The plug-in includes algorithms for the individuation of fault locations based on fault types and constraints discussed in Section 3.4. The injection consists of parsing the target binary code to individuate assembly instruction sequences corresponding to programming constructs, i.e. function calls, assignments and selection statements. Once patterns are recognized, modifications are applied by substituting assembly instructions with a *nop* instructions, performing statement removal.

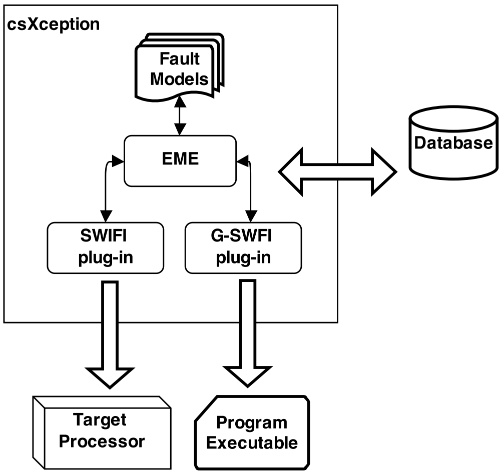


Figure 3.19: csXception^{TM} architecture

3.4.4 Test Planning

In this section, technical details about test-suites generated by FaultProg are discussed. As shown in 3.7, the generator takes several input parameters that determine the number and the content of synthetic programs. In total, 4,855 synthetic programs were generated considering both respected (that are 1,984) and violated constraints (that are 2871). 3.8 reports the number of generated programs divided by fault types. It can be noticed that, for each combination of fault-type/constraints, the number of programs changes among the fault types. In fact, some inputs combinations are not valid and programs are not generated. For instance, for MLAC fault type that removes an AND in the expression of if-condition, it does not make sense to generate a program in which the if-condition contains a simple variable or a function call. These considerations contributes to lower the number of tests to perform in order to individuate inaccuracies.

3.4.5 Results

In total, 3,429 source-level faults and 1,562 binary-level faults were generated, respectively. Their distribution across fault types is shown in Figure 3.20. The two distributions exhibit that more source-level faults are injected with respect to the majority of fault types, whereas in other cases more binary-level faults are injected (such as MFC and MIEB). The comparisons between source-level and binary level faults generates Correctly Injected, Spurious and Omitted faults as shown in Figure 3.21. Causes of spurious and omitted faults are then

Table 3.8: Test-suites generated by FaultProg

Fault Type	**Programs per Constraint**	
MFC	C00	240
	C01	80
	C02	240
MVA	C00	10
	C02	135
	C03	150
	C04	150
	C05	90
	C06	45
	C07	150
MIA	C00	240
	C08	240
	C09	240
MIFS	C00	240
	C02	225
	C08	240
	C09	240
MIEB	C00	240
	C09	240
MLAC	C00	192
MLOC	C00	192
MLPA	C00	240
	C01	240
	C02	240
WVAV	C00	16
	C03	16
	C04	16
	C06	3
WPFV	C00	32
	C01	10
	C03	32
WAEP	C00	192

analysed.

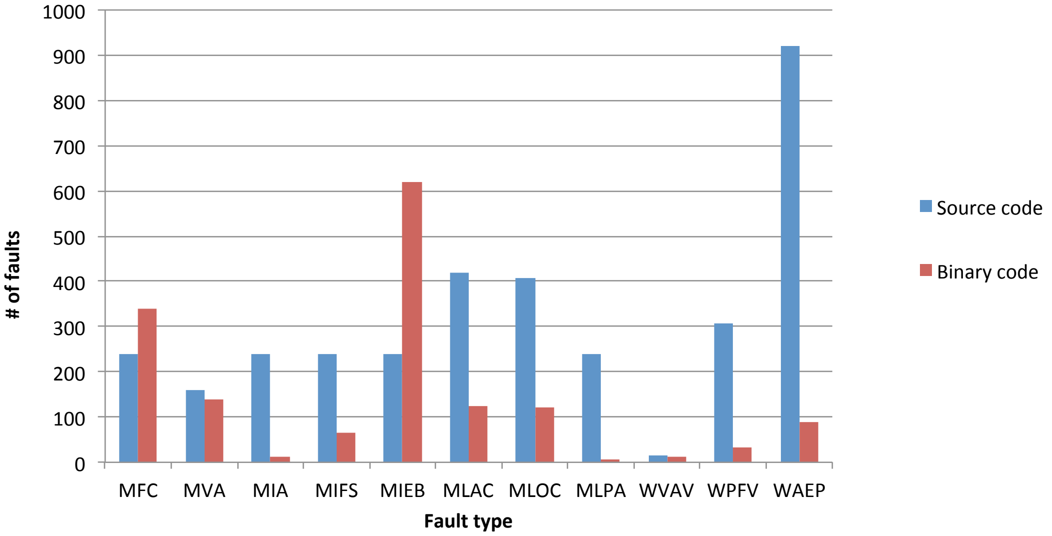


Figure 3.20: Distributions of Faults Injected at Binary and Source Code Level

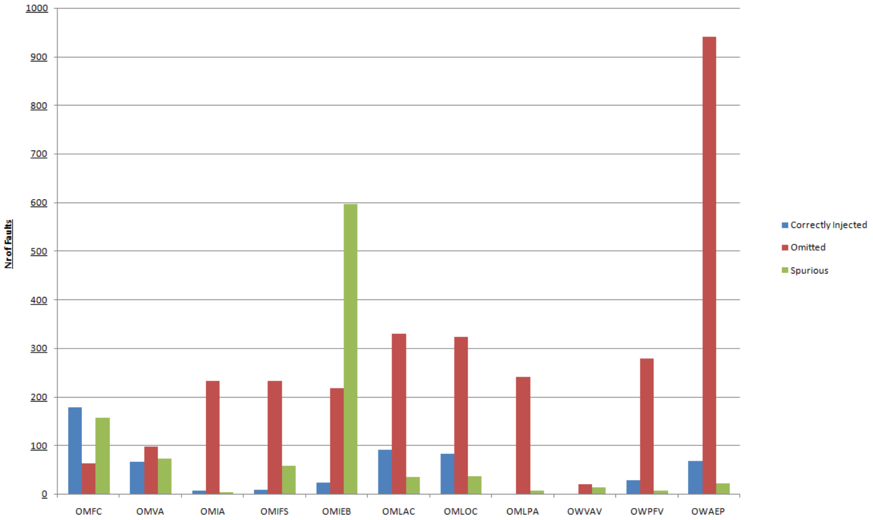
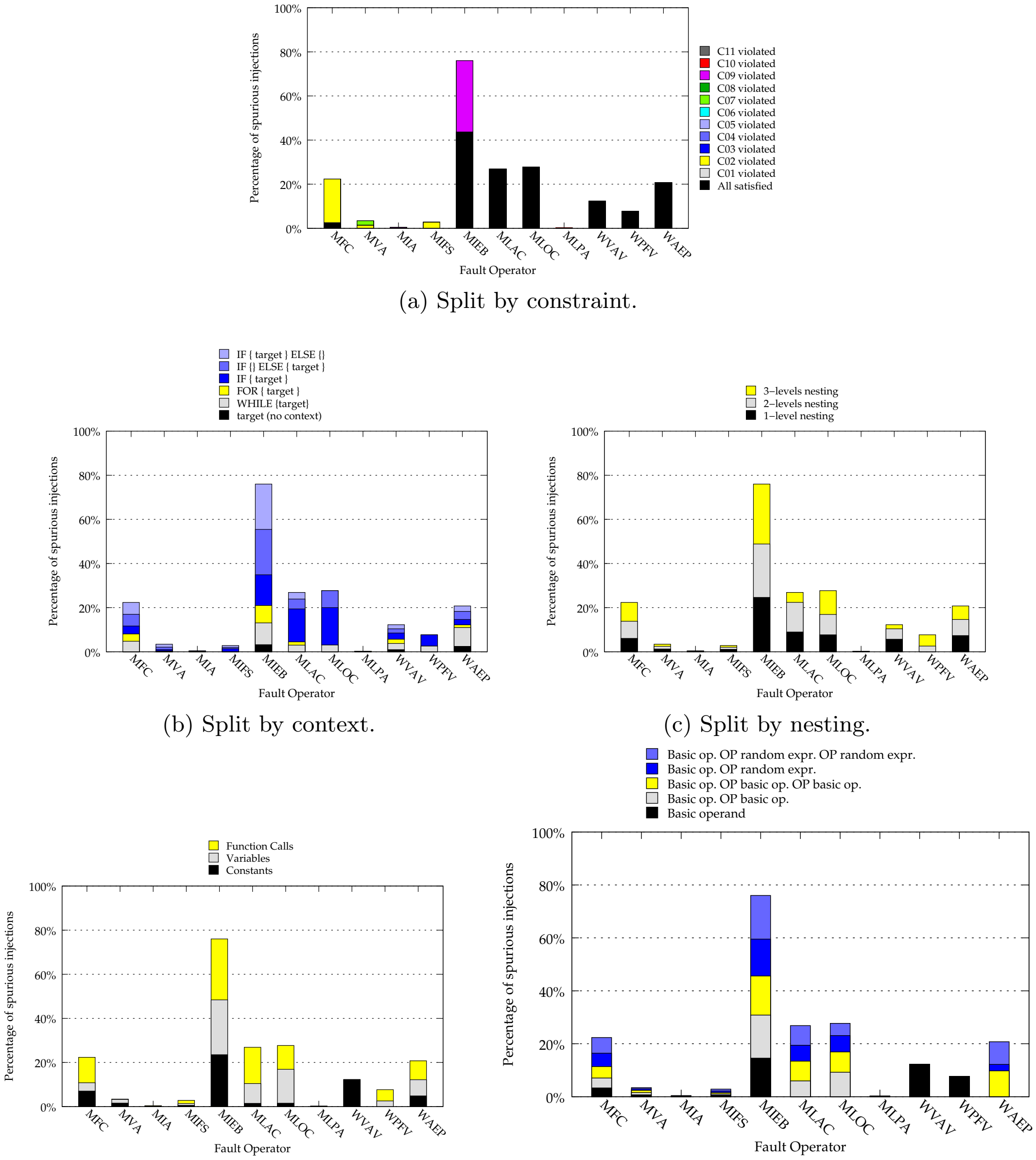


Figure 3.21: Distributions of Correctly Injected, Spurious and Omitted Faults.

Plots in Figure 3.22 show the percentage of programs, grouped by fault type, for which csXception produces spurious faults. Then, programs containing a spurious fault are grouped by specific parameters such as constraints (Figure 3.22a), type of context that surrounds the target fault location (Figure 3.22b) Firstly, it emerges that the higher number of spurious faults is obtained for MIEB, MFC, MLAC, MLOC fault types. Looking at Figure 3.22a, most of the MIEB and MFC spurious faults is obtained when in synthetic programw where constraints C09 and C02 are violated. This highlights that the implementation of these constraints could be improved. An example of spurious fault is depicted in Figure 3.23. The function call *func_ 7()* is the only instruction in a while block violating the constraint C02: according to Table 3.6 the injection should not be performed. The reason of the wrong injection is probably due to the complexity of the input expression of *func_ 7()* that was erroneously interpreted as additional statements of the while block. An improvement to the algorithm that counts the number of statements in a block could avoid these spurious injections. For instance, in the example of Figure 3.23, the tool should recognize that the result of two expression is the input parameter of the function. We found a similar problem for the MIEB fault type, since the tool does not correctly computes the number of statements within an if block, thus leading to spurious injections when an if block contains a loop or more than 5 statements. As for MLAC and MLOC, spurious injections cannot be attributed to the erroneous implementation of the constraints since they are not associate to any constraints. Instead, they depend on the complexity of the target location (only non-black bands appear in the bars of MLAC and MLOC) as it is observed in Figure 3.22e. So that, an hypothesis can be formulated: the complexity of boolean condition influences the presence of spurious injections. Analyzing some of these violations confirmed the hypothesis: we found that the tool injects a spurious MLAC/MLOC when the if construct contains a boolean condition with three or more logical clauses.

A similar issue was found for WAEP fault type: when an input parameter of the function call is a complex expression, the tool performs a number of injections greater than it would be allowed by the fault type definition.

Regarding omitted faults Figure3.24, the percentage of omitted faults is very high for most of fault type, i.e all the synthetic programs lead to at least one omitted injection in the target fault location. The great number of omitted faults confirms the results of the experimental evaluation of binary fault injection (see Section 3.3) that demonstrated the omitted injections are very frequent. Because of the high number of the omitted faults for each fault type, it is not possible to find Analyzing Figure 3.24, it emerges that the nesting, the operands and the complexity of the expressions do not influence significantly the omitted injections, i.e. there is not a specific value of nesting, operands, or complexity that causes omitted injections: the bands in the bars of Figure 3.24a, Figure 3.24b Figure 3.24c have similar widths in almost all cases, so it is unlikely that omitted injections are caused by



(a) Split by constraint.

(b) Split by context.

(c) Split by nesting.

(d) Split by type of operand.

(e) Split by complexity of expressions.

Figure 3.22: Spurious injections.

```
void func(void) {

    ...

    while ((l_6 & func_7(g_139))) {

        // start target fault location
        func_7((0x34552768L && func_13((g_4 % mod_rhs(0x7790L)), (g_4 % mod_rhs(1L)))));
        // end target fault location
    }

    ...

}
```

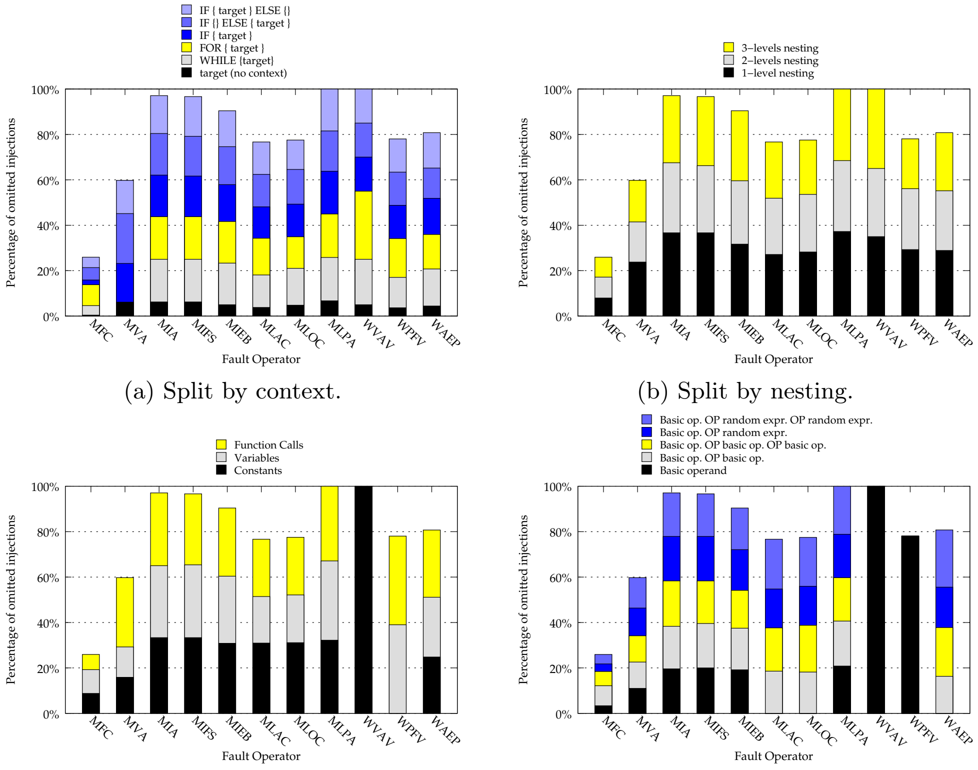
Figure 3.23: Example of synthetic program causing a spurious injection.

specific value of these parameters. Instead, from Figure 3.24a it seems that omitted injections tend to occur when the target fault location is included within a "context" construct, such as a for/while loop or an if construct. Thus, omitted injections can occur because the tool does not correctly discriminate between the target statement and the context construct. For instance, Figure 3.25 depicts part of a synthetic program in which MFIS fault is omitted, i.e. the fault is not injected even if the target fault location is a valid injection point.

In cases like this, it seems that the tool is confused by programs with complex control flows, and is not able to analyze statements that are nested within some conditional construct.

3.5 Summary

In this chapter, it is evaluated the accuracy of a binary software fault injection technique (G-SWFIT) that injects faults in the binary code of a program. The accuracy of faults injected at binary level has been assessed by comparing the faults injected in the source



(a) Split by context.

(b) Split by nesting.

(c) Split by type of operand.

(d) Split by complexity of expressions.

Figure 3.24: Omitted injections.

```
...
while (g_101) {

    int8_t l_146 = 0x6FL;

    while ((func_46(g_109, lshift_s_s(g_78, func_56(l_144, g_145, g_127, g_83, g_112)))
    & (((l_146 % mod_rhs(g_81)) || (g_147 | g_127)) == g_131))) {

        uint32_t l_150 = 0xD765C340L;
        int8_t l_155 = 0L;

        while ((g_148 & func_56(func_44(g_89), l_149, (g_82 - g_99), (g_130 & g_101))){

            int8_t l_153 = -1L;
            g_51 = l_150;
            int8_t l_152 = 0xF3L;

            // start target fault location
            if ((rshift_s_s(g_80, g_4) | (g_78 ^ 0L))) {
                g_147 = g_92;
                func_3((g_105 + g_113));
                g_58 = (g_109 <= g_90);
                g_17 = g_151;
                func_3(l_152);
            }
            // end target fault location

            ...

        }
    }
}
...
```

Figure 3.25: Example of synthetic program causing an omitted injection.

code by using the same fault injection rules. The analysis pointed out improvements to both tools involved in the comparison. Results can be summarized as follows:

- The accurate injection of software faults in the binary code is challenging in complex software systems. A large number of omitted and spurious faults was observed in the first analysis: for each injected fault there is about 1 omitted fault that has not been injected, and about half of the injected faults were spurious. Moreover, the problem is more significant where the code complexity is greater, as in the case of application-level code in the case study.

- Several omitted and spurious faults are due to the lack of high-level information in the binary code, and most of them are due to macros and inline functions. These inaccuracies have to be accepted as limitations of fault injection at binary level, and should be taken into account when conclusions are drawn from fault injection experiments. In some cases, such limitations can be considered acceptable: for instance, when the aim of fault injection is a coarse-grained analysis of failure modes (e.g., the relative percentage of crashes or stalls of the system), the results can be adequately estimated even in the presence of inaccurate injected faults [36, 60]. Instead, fault injection at the source level is advisable when the source code is available and a more fine-grained analysis of the effects of injected faults on the system is needed.

- Several omitted and spurious faults are not related to limitations of fault injection at binary level, but they are due to the incomplete or simplified implementation of G-SWFIT. In particular, issues are related to the implementation of fault type constraints and to the identification of code blocks and control structures. These issues are not due to the G-SWFIT technique, and they can be avoided if an experimental evaluation of the fault injection tool is performed to improve the implementation. If these aspects are improved, then omitted and spurious faults represent the minority of cases. A future research work consists in extending the proposed method in order to support the development of SFI tools at binary level, since such tools need to be re-engineered or developed from scratch when fault injection is performed in a new hardware architecture or in a system adopting a different compiler. In this context, faults injected at the source code level can be potentially exploited to understand how software faults are translated in binary code and how fault operators can be implemented.

Chapter 4

Achieving Representativeness in Interface Error Injection

4.1 Introduction

The integration of OTS components plays a key role in the development of software systems. Unfortunately, component-based software development imposes significant risks for dependability [37,115,116]: When a component is reused in a new context, the system may use parts of the component that were previously seldom used and only lightly tested, or may interact with the component in unforeseen ways, thus exposing residual software faults in the component that had not been discovered before.

Despite the extensive development of various approaches, SFI remains a complex process, and technical limitations affect the feasibility and the quality of SFI experiments. As discussed in the previous chapter, the mutation of components' code requires the ability to mutate the binary code of the OTS component. This was has proven very difficult: In some cases it is impossible to correctly recognize and mutate high-level programming constructs in binary code [29]. Another issue with CM is *efficiency*, in terms of number of experiments that actually exhibit a component error, since injected faults may be difficult to activate and not produce any perceived error during the experiments [25]. *Interface error injection* (IEI) is an alternative SFI approach that overcomes these limitations of CM. The representativeness of interface errors is less of an issue for traditional testing, where invalid values are useful at exposing inputs that lead to software failures. Nevertheless, the use of IEI for the representative emulation of component faults (as required by dependability assessment strategies [62,78,115]) is questionable, as there is a lack of evidence that IEI can realistically emulate software faults.

This chapter aims at analysing how software faults in components' code result in errors at components' interfaces, in order to provide some constructive evidence towards more representative IEI techniques.

The chapter is structured as follows. After a discussion of related work on the relation between software component faults and interface errors in Section 4.2, Section 4.3 introduces our system model and identifies possible locations for inter-component error propagation. 4.3.1 shows how this information can be exploited to design an approach for the analysis of inter-component error propagation. In Section 4.3.2, it is discussed the general intra-component FI process, the details of the experimental setup, and the obtained results. Then, it is discussed the implications of our findings for the construction of representative interface error models in 4.4.

4.2 Background and Related Work

The representativeness of faults is a key property for the *quantitative assessment* of dependability properties through fault injection. In [84], Ng and Chen designed a write-back file cache with the requirement to be as reliable as a write-through file cache. To validate this requirement, software faults are injected in the OS to estimate the probability of data loss. Using fault injection experiments, the authors identified weak points of their file cache and iteratively improved its design until its reliability was comparable to a write-through cache. In [23], fault injection was adopted to evaluate whether the PostgreSQL DBMS exhibits fail-stop behavior in the presence of software faults, and to measure its fault detection latency. The study found that the transaction mechanism is effective at preventing fail-stop violations, reducing them from 7% to 2%. Kao et al. [64] performed a Markov reward analysis, based on fault injection experiments, to quantify the expected impact of faults on performance and availability. Tang and Hetch [109] proposed an approach for accelerating the probabilistic evaluation of high-reliability systems (e.g., with a failure rate in the order of 10^{-6}) that adopts fault injection to force the occurrence of *rare events*. In [114], Voas *et al.* inject errors within a program to identify where to place assertions and to avoid error propagation. The accuracy of these measures and the confidence on fault tolerance mechanisms is based on the assumption that the injected faults are representative of real software faults. In [112], Vieira and Madeira proposed a dependability benchmark to evaluate different DBMS configurations with respect to operator and software faults in order to aid system administrators; in this case, a representative set of faults is required to make systems comparable and to identify the best configuration.

The representativeness of error injection techniques with respect to software faults was investigated in many studies. In order to accelerate the consequences of software fault injection experiments through error injection, Christmansson and Chillarege [25] proposed a methodology to derive a set of representative errors that match the effects of residual software faults of a system, by analyzing failure data at the users' site. They proposed to inject errors through bit-flipping, which corrupts program data at run-time by changing the contents of individual bits or bytes on heap, global, and stack areas, and mechanisms that were originally developed for emulating the effects of hardware faults [15, 61]. The error types were derived as the *immediate effect* of fault activations on *internal program data* and classified according to the type of data corrupted by the fault (e.g., corruption of address vs. data words). Christmansson et al. [26] observed the benefits of such error injections over fault injections for evaluating the fault-tolerance of an embedded real-time system in terms of experiment setup and execution time. Their experimental analysis also showed that the lack of error representativeness has a noticeable impact on experimental results.

It must be noted that the approach of [25] can emulate the effects of software faults only to a limited extent, as Madeira et al. [72] showed that bit-flipping is not suitable for mimicking faults that involve several statements and complex data structures. Instead, Daran and Thévenod-Fosse [30] showed that code mutations are effective at emulating software faults, by observing an overlap of the error propagation of 12 known real faults and 24 mutations in a small safety-critical program. Nevertheless, their analysis focused on *internal errors* rather than *interface errors.*

To analyse how faulty components can affect other components, the focus is on *error manifestations at component interfaces*, rather than immediate effects on internal data of the targeted component as in [25,30]. Moraes et al. [77] and Jarboui et al. [58] investigated the representativeness of error injections at component interfaces, by comparing the failure distributions obtained from IEI and from CM, respectively. From a series of comparative experiments between fault injection based on representative code changes [36] and data-type-based interface errors commonly adopted in robustness testing (encompassing parameter corruptions through bit-flipping, boundary values such as -2^{31}, and invalid values such as NULL pointers [67,118]), they concluded that IEI and CM produce failures.

A limitation of previous analyses on error propagation [30,58,77] was that they were manually performed on a very small number of faults and on single programs, due to the lack of an automated tool for analyzing interface error propagation. Our study thus proposes an automated approach able to analyze arbitrary memory corruptions of component interface data, focusing on data exchanged via inter-component interfaces. Unlike previous tools for error propagation analysis by Kao et al. [64] and by Chandra and Chen [23], our method is able to precisely distinguish between the corruption of internal component data and of interface data.

The experimental analysis aims at identifying how *software faults* in a software component turn into *interface errors* that affect other components and the system as a whole. Figure 4.1 depicts the relationship between faults and errors: When a *component service* of the *target* component is requested through the *component interface* (e.g., through an API function call) by a *user* component, the target processes input data from the user, and provides results, by manipulating *interface parameters* provided by and returned to the user (e.g., data structures exchanged through input/output parameters and through the return value of a function invocation). During the execution of a component service, the activation of residual software faults in the component results in an *internal error*, e.g., the corruption of internal data or a change of the control flow. When the component service terminates, the interface parameters exchanged between the target and the user components can be corrupted as an effect of such internal errors, thus producing *interface errors*. In such cases, errors *propagate* from the target component to other components.

Software components are considered in the form of *libraries* (i.e., collections of functions

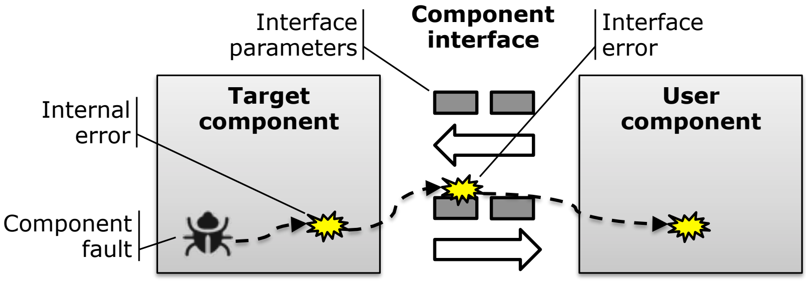
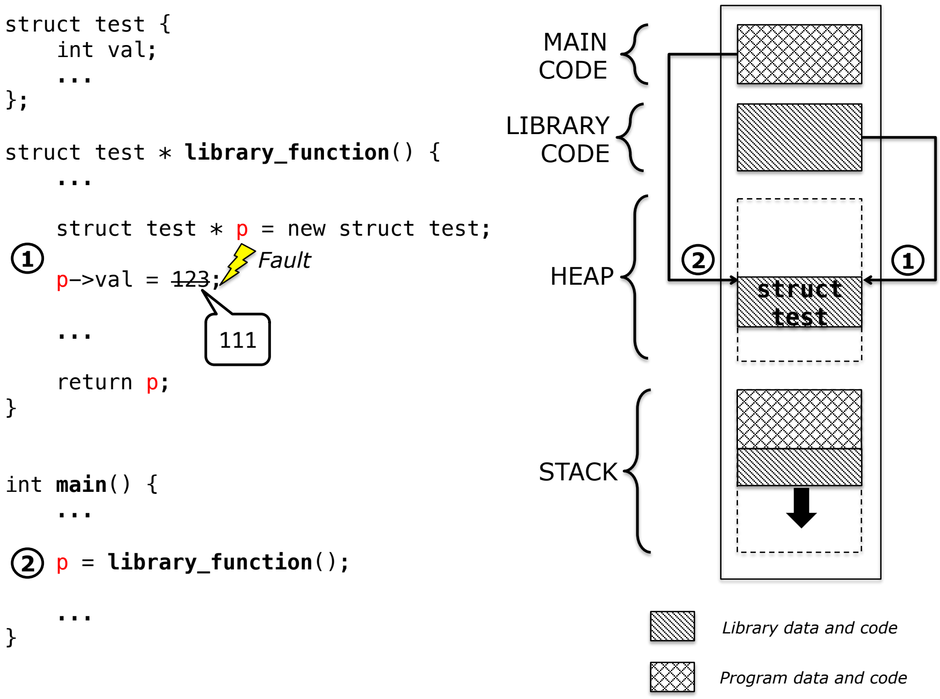


Figure 4.1: Relationship between component faults and interface errors.

and classes) linked to a C/C++ main program at compile- or at run-time, as these languages are predominant in safety-critical control systems and systems software. However, the general approach applies for any type of software composition where components exchange data through shared data structures. Figure 4.2 to Figure 4.5 show the resulting error propagation paths for data errors in the case of library functions invoked from a (*main*) program.

The first scenario (Figure 4.2) consists in the corruption of a data structure that is dynamically allocated on the heap by the library , where the corrupted data structure survives the component invocation and is returned to the main program through a pointer return value (either on the stack or in a register, depending on calling conventions), which represents an erroneous interface parameter.

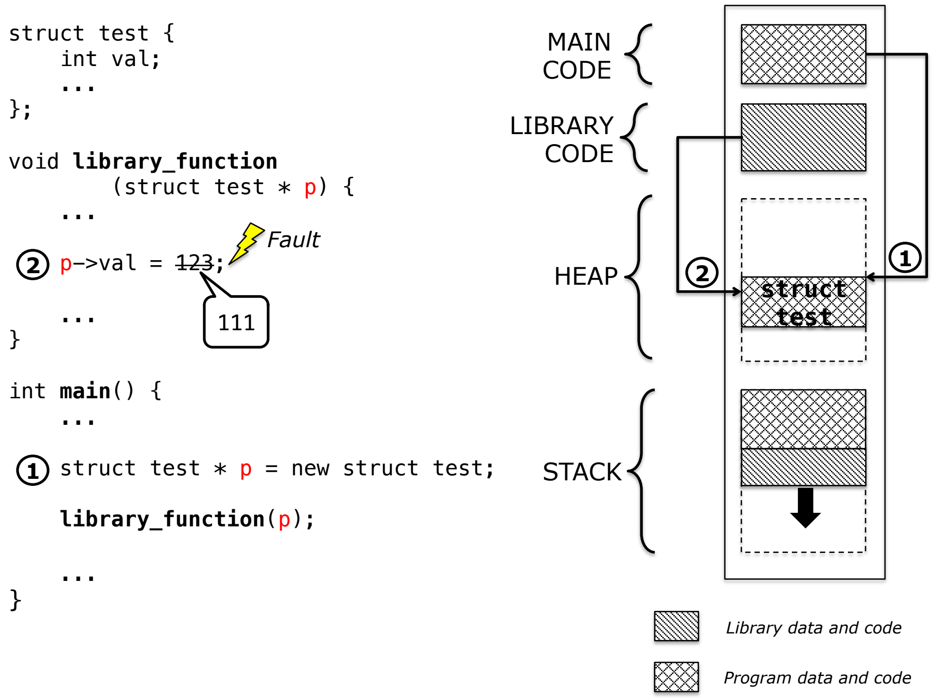
Figure 4.3 depicts a similar case, in which a data structure is allocated by the main program , passed to the library through a pointer interface parameter, and corrupted during the library invocation



```
struct test {
    int val;
    ...
};

struct test * library_function() {
    ...

    struct test * p = new struct test;

①   p->val = ~~123~~;

    ...

    return p;
}

int main() {
    ...

②   p = library_function();

    ...
}
```

Figure 4.2: Propagation through a library-allocated heap area.

Even if interface parameters are not directly corrupted, error propagation can still *indirectly* affect the main program by corrupting data that is pointed to by an interface parameter, such as in the case of complex data structures like trees and linked lists. This is the case in Figure 4.4, where a user-allocated data structure is linked to a library-allocated string that can get corrupted. A corruption of the linked string can be considered an interface error, as this area is reachable by the main program. This applies in general to any memory area reachable from an interface parameter through an arbitrary number of pointers.

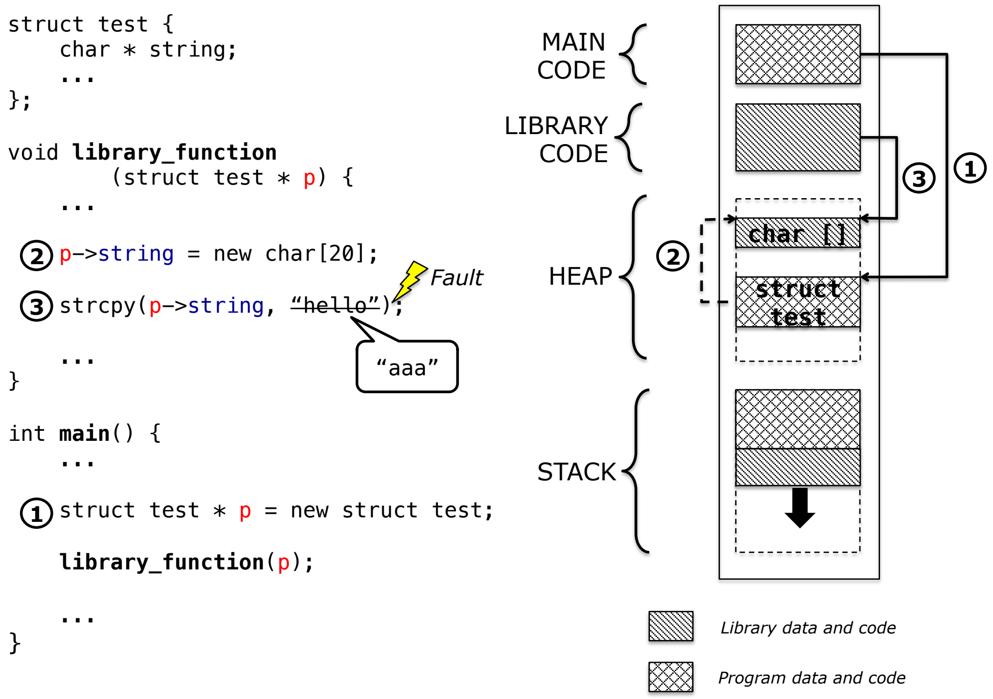


```
struct test {
    int val;
    ...
};

void library_function
        (struct test * p) {
    ...

② p->val = ~~123~~;
                111
    ...
}

int main() {
    ...

① struct test * p = new struct test;

    library_function(p);

    ...
}
```

Figure 4.3: Propagation through a user-allocated heap area.

4.3 Propagation of Errors at Component Interfaces

Finally, error propagation is not limited to heap areas, as in the case of Figure 4.5, in which an array is allocated as a local variable by the main program , a pointer to the array is passed to the library through an interface parameter, and the array's contents gets corrupted by the library It is important to note that the analysis do *not* involves internal errors that are not visible to the main program (i.e., memory areas not reachable outside the component). This is the case, for instance, of local variables allocated by the library, and of heap memory areas not reachable (neither directly through interface parameters, nor indirectly) by the



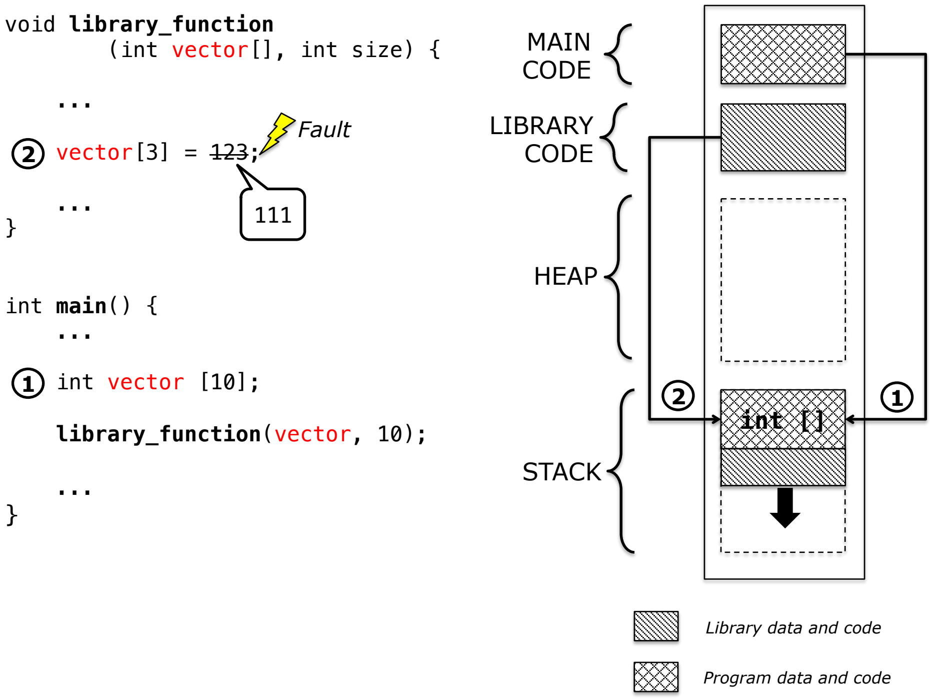
```
struct test {
    char * string;
    ...
};

void library_function
        (struct test * p) {
    ...

 ②p->string = new char[20];

 ③strcpy(p->string, "hello");

    ...
}

int main() {
    ...

 ①struct test * p = new struct test;

    library_function(p);

    ...
}
```

Figure 4.4: Propagation through a library-allocated heap area, reached through a user-allocated heap area.

main program.

4.3.1 Propagation analysis approach

The proposed method enables the automated analysis of errors occurring at the interfaces of C/C++ software components, according to the workflow of Figure 4.6. First, the library is linked to a main program (which represents the *workload* of the experiment) and executed, collecting information about (i) memory *stores* made by the library, (ii) dynamic memory allocations of both library and main program, and (iii) library invocations performed by the program during the execution. The raw execution trace is pre-processed, in order to identify



```
void library_function
        (int vector[], int size) {
    ...
② vector[3] = ~~123~~;
    ...
}

int main() {
    ...
① int vector [10];
    library_function(vector, 10);
    ...
}
```

Figure 4.5: Propagation through a user-allocated local variable.

library memory *stores* that affect memory areas actually visible to the main program (such as the cases considered in Figure 4.2 to Figure 4.5). The same steps are performed a second time, with a software fault deliberately injected into the library. Due to the injected fault, the library can generate different memory *stores* to interface parameter data, which leads to interface errors. To identify such interface errors, two execution traces are compared and differences are pointed out in terms of memory *stores* that write incorrect data (i.e., values differing from the fault-free execution), memory *stores* omitted by the faulty library, and superfluous memory *stores* that are only performed in the faulty execution.

To trace memory accesses performed by the target library, we perform a *dynamic binary*

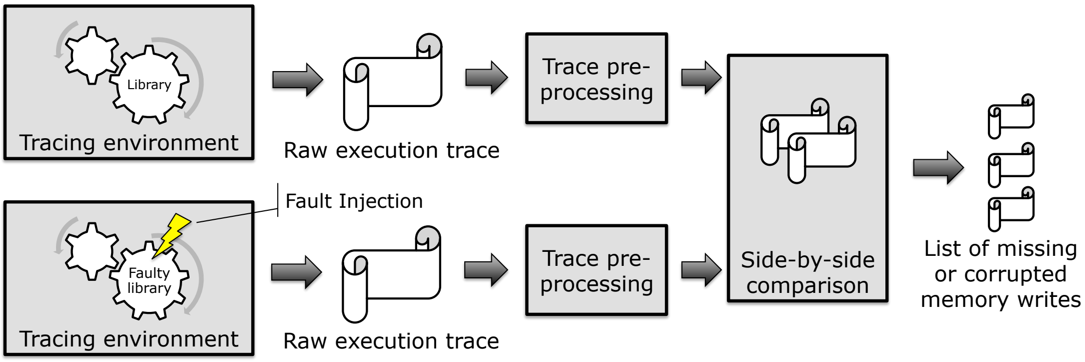


Figure 4.6: Overview of error propagation analysis.

instrumentation (DBI) of the executable program [83]. In general, DBI techniques instrument a program during its execution by adding *analysis code* that collects data about the state of the execution. Uses of DBI range from simple analyses, such as profiling of function calls and code coverage, to more complex analyses, such as undefinedness of program variables. In particular, we adopt the *disassemble-and-resynthesise* DBI approach [83], which translates the original program (native code) into an *intermediate representation* (IR), instruments the IR, and translates the IR back to native code, which is then executed on the native system. The IR code consists of architecture-independent, RISC-like instructions that perform individual operations such as memory stores (in contrast to a native CISC-like code instruction, such as x86 instructions, that can have several side effects). DBI takes advantage of conventional compiler optimizations, such as code caching, in order to accelerate the process of instrumentation. Analysis code is mixed with the original IR code to obtain an instrumented IR code: for instance, to track memory modifications, the DBI can add one or more IR instructions after each IR *store* instruction, to record the accessed address and the value written to that address. This approach allows to analyze memory accesses made by a program at a fine grain, which is the objective of the analysis of this study.

We developed a DBI analysis tool for tracing library code on top of the Valgrind program analysis framework [83]. Our tool inserts the following analysis code at run-time:

- After each instruction, we insert code to check the instruction address to detect whether the control flow moved from the main program to the code of the target library (i.e., the program enters in *library context*). In a similar way, we check whether the control flow returns from the library to the main program. We record the name of the invoked library function (which is obtained from the symbol table included in the library), and the return value of the library invocation.

- When library context is entered, we record the current value of the stack register, which marks the end of the stack frame of the main program (containing local variables of the main program) and the beginning of the stack frame of the library (containing local variables of the library). While the execution is in library context, we record changes to the stack register, in order to trace the growth of the library stack frame and, ultimately, to identify writes to local variables of the main program (which are stored on the stack) and to discard writes to local variables allocated by the library.

- While in library context, after each IR *store* instruction, we insert code for recording the address of the instruction that writes to memory, the address and the size of the area being written, and the new contents of the memory area. The DBI tool records memory accesses to heap and global data (e.g., Figure 4.2, 4.3, 4.4), and to data in the stack frame of the main program (e.g., Figure 4.5).

Moreover, the tool wraps and intercepts the invocation of the following functions of the C library:

- Invocations of `mmap()`, which is invoked at run-time by the loader to link a shared library to the address space of the process: We record the addresses of memory areas in which library code and data are mapped.

- Invocations of memory allocation functions (e.g., `new`, `malloc()`), both in library context and in the main program: We record the address and the size of each allocated and freed memory area, and the code location that allocated that memory area. This information is used later in the analysis for identifying memory areas reachable by the main program.

As a result, the execution trace obtained from the DBI tool provides (i) all invocations of and returns from library functions (LIB_INVOCATION and LIB_RETURN events), and their return value, (ii) all memory writes made by the library outside its local variables (STORE events), and (iii) all memory allocations and deallocations (ALLOCATION and FREE events).

The trace is then processed (Figure 4.6) to identify memory *stores* that *write data accessible by the main program*, that is, interface parameter data. These data are identified by building a *graph* where nodes represent *memory areas* (i.e., a range of contiguous memory addresses, such as an array of bytes allocated on the heap), and edges represent *pointer-pointee* relationship between memory areas (i.e., a memory area contains a pointer variable, pointing to another memory area). A memory area is *reachable* by a program using the library if there is a path in the graph between that memory area and any variable of the user program, i.e., a variable in the user heap (represented by the *UH* node), in the user stack (*US* node) or an output value from the library function call (*O* node). Figure 4.7 shows an example.

```
struct test { char * s; };

int main() {
   int v [10];
   struct test * p = new struct test;

   char * c = library_function(p, v);
}

char * library_function
          (struct test * p, int v[]) {

   char * tmp = new char[3];
   delete [] tmp;

   p->s = new char[20];

   return new char[5];
}
```

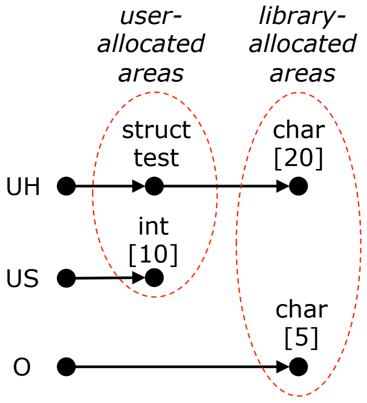


Figure 4.7: Example of reachability graph.

4.8 provides the detailed algorithm for building and analyzing the graph. The trace is analyzed in three passes. Each pass processes events of the trace in sequential order, by invoking for each event a function according to the type of event (e.g., when a STORE event is encountered while scanning the trace, it is processed by invoking HANDLE_STORE).

Pass 1. This pass (Figure 4.8a) identifies heap memory areas that are allocated and de-allocated within the same library invocation (i.e., “temporary” memory areas used during an individual library invocation, such as “tmp” in Figure 4.7), and removes them from the analysis (Fig. 4.8a, line 20), since these areas cannot be accessed by the user program (they do not “survive” a library invocation). The remaining heap areas can still potentially be accessed by the main program. The *address ranges* $[start, end)$ of remaining heap areas are arranged in an *interval tree*, the *Allocs* set (Fig.4.8a, line 8), which is a data structure that allows to search for ranges containing a given value: We use this feature in subsequent passes to find, for a given address in the trace, the heap area to which that address belongs. Address ranges of global data structures of the library (obtained from the library symbol table) are also inserted in the interval tree at the beginning of the pass (Fig. 4.8a, line 1

Pass 2. This pass (Figure 4.8b) constructs a directed graph (E, V) representing pointer-pointee relationships between li- brary-allocated memory areas, and between these areas and memory areas of the user program. Node A of the graph (representing a memory area A) is connected to node B if the area A contains a pointer with an address to the memory area B (i.e., B is “reachable” by A). The graph includes a node for each library-allocated memory

(a) Pass 1: Collection of memory allocations.

```
 1: $E \leftarrow$ get_library_allocated_areas($Allocs$) $\cup \{UH, US, O\}$
 2: $V \leftarrow \emptyset$

 3: function handle_store($store$)
 4:     $pointed\_area \leftarrow$ IntervalSearch($Allocs, store.value$)

 5:     if $pointed\_area \neq \emptyset \wedge$ (is_lib_heap_area($pointed\_area$) $\vee$ is_lib_global_area($pointed\_area$)) then

 6:         $accessed\_area \leftarrow$ IntervalSearch($Allocs, store.address$)
 7:         if $accessed\_area \neq \emptyset$ then
 8:             if is_user_heap_area($accessed\_area$) then
 9:                 $V \leftarrow V \cup (pointed\_area, UH)$
10:             else
11:                 $V \leftarrow V \cup (pointed\_area, accessed\_area)$
12:             end if
13:         else if is_user_stack_data($store.address$) then
14:             $V \leftarrow V \cup (pointed\_area, US)$
15:         end if
16:     end if
17: end function

18: function handle_lib_return($returned\_value$)
19:     $pointed\_area \leftarrow$ IntervalSearch($Allocs, returned\_value$)
20:     if $pointed\_area \neq \emptyset$ then
21:         $V \leftarrow V \cup (pointed\_area, O)$
22:     end if
23: end function
```

(b) Pass 2: Generation of the reachability graph.

```
 1: $Trace \leftarrow \emptyset$

 2: function handle_store($store$)
 3:     $area \leftarrow$ IntervalSearch($Allocs, store.address$)
 4:     $address \leftarrow store.address$
 5:     if is_lib_heap_area($area$) $\vee$ is_lib_global_area($area$) then
 6:         if is_reachable_by_user($V, E, address$) then
 7:             $Trace \leftarrow Trace \cup \{store\}$
 8:         end if
 9:     else if is_user_heap_area($area$) $\vee$ is_user_stack_data($address$) then
10:         $Trace \leftarrow Trace \cup \{store\}$
11:     end if
12: end function
```

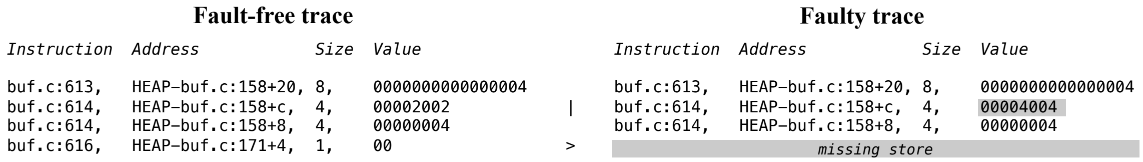
(c) Pass 3: Event filtering.

Figure 4.8: Trace pre-processing.

area. Moreover, we introduce in the graph the UH, US, and O nodes (Fig. 4.8b, line 1): if a node A is connected to any of these nodes, then the memory area A is directly reachable through user-allocated heap memory, user stack memory, or an output value of a function call, respectively. To identify pointer-pointee relationships, we check the value written by *store* operations (*store.value*) and see whether that value represents an address within one of the memory areas in *Allocs*: if this is the case, then the written value represents a pointer, and the two areas (i.e., the one containing the pointer, and the one with the pointed address) are connected in the graph (Fig. 4.8b, line 11). If a library-allocated heap/global area is pointed to by user heap areas, the user stack, or values returned by library invocations, that library-allocated area is connected to UH, US, or O, respectively (Fig. 4.8b, lines 9, 14, 21) . This pass uses an interval tree search in *Allocs* (Fig. 4.8b, line 4) to identify pointers and the areas they point to.

Pass 3. It identifies memory *stores* to areas that are reachable by the main program (Figure 4.8c). If the address of the *store* (*store.address*) belongs to a library-allocated area, the algorithm inspects the graph using the IS_REACHABLE_BY_USER function (Fig. 4.8c, line 6) to find whether the area is reachable outside the library, and only adds the *store* to the final trace if there exists a path in the graph between the memory area and one of the US, UH, or O nodes (i.e., the area is reachable by the user). *Stores* on user-allocated memory are also included in the trace (Fig. 4.8c, line 10).

After the execution of an experiment and of pre-processing, we obtain a trace consisting



```
                Fault-free trace                                                      Faulty trace
Instruction  Address             Size  Value                     Instruction  Address             Size  Value

buf.c:613,   HEAP-buf.c:158+20, 8,    0000000000000004           buf.c:613,   HEAP-buf.c:158+20, 8,    0000000000000004
buf.c:614,   HEAP-buf.c:158+c,  4,    00002002              |    buf.c:614,   HEAP-buf.c:158+c,  4,    00004004
buf.c:614,   HEAP-buf.c:158+8,  4,    00000004                   buf.c:614,   HEAP-buf.c:158+8,  4,    00000004
buf.c:616,   HEAP-buf.c:171+4,  1,    00                    >                        missing store
```

Figure 4.9: Example of comparison between faulty and fault-free traces.

of a sequence of tuples, each representing a memory *store* performed by the library on user-reachable memory. A tuple is defined as: <*instruction address*, *memory address*, *store size*, *stored value*>. A "faulty" execution trace is then compared with a "fault-free" execution trace. Given that execution traces are always identical when the target software is executed without faults (effects of non-determinism must be factored out, as discussed below), any differences between the faulty and the fault-free traces are actually due to injected faults. Traces are compared by searching for the *longest common subsequences*, using the algorithm described in [55]: it aligns two sequences such that two tuples at the same position in the aligned sequences will have the same values, by comparing, respectively, the instruction, the address, the size and the value of memory *stores*. In the example of Figure 4.9, the first and the third *stores* of both sequences are aligned; the *stores* at the second position are performed by the same instruction on the same memory area (a heap area allocated at *buf.c:158*), but a wrong value is written in the faulty execution; the fourth *store* is only performed in the fault-free execution, while it is omitted in the faulty one. In this example, 4 bytes are corrupted by writing a wrong value at the second position, and another byte is corrupted since its initialization is omitted at the fourth position. We also detect corruptions due to spurious *stores* not performed in the fault-free execution. In a similar way, we compare *return values* of library invocations.

It is important to note that, when comparing faulty and fault-free traces, we focus on memory *stores* and return values produced by *the first library function invocation that exhibits differences from fault-free executions.* We do so since differences exhibited by subsequent invocations of library functions may not be due to the injected fault, but due an incorrect behavior by the main program, caused by the effects of the first "faulty" library invocation. Focusing on the first faulty library invocation avoids confusion between effects and provides a more precise evaluation of interface errors.

Another important aspect that we needed to take into account in the design of our DBI technique is the degree of *non-determinism* in execution traces. The comparison of traces in faulty and fault-free conditions (as depicted in Figure 4.6) requires that differences between traces are actually due to faults, and not due to random variations caused by non-determinism. In our experimental setup, we took into account the following sources of non-determinism:

Memory management. A dynamically-allocated memory area can be mapped at different addresses in different executions. To enable the comparison of *store* operations performed on the same heap area, we rewrite memory addresses in the trace by replacing *absolute addresses* of heap memory areas with *relative addresses* within that area. Relative addresses are composed by a pair <*area id*, *offset*> (e.g., Figure 4.9); the *offset* represents the distance between the beginning of the heap area and the address being rewritten, and the *area id* is a number that uniquely identifies the allocation, which is computed from the code location where the area was allocated, the call stack at the time of allocation, and an incrementing integer. This allows to identify two identical *stores* (i.e., *stores* performed by the same instruction, on same heap area, and with the same value) even if the heap area is mapped at different addresses. In a similar way, the trace is rewritten to replace addresses belonging to global areas with relative addresses.

Thread scheduling. The program execution flow and, therefore, the sequence of *stores* performed during the execution, can vary among executions due to thread scheduling. *Recording and replay* techniques can be adopted to mitigate this source of non-determinism [88, 108]: the reference execution (i.e., the execution without faults) can be recorded, and then replayed while executing the faulty version of the target software. Since the current implementation of our DBI tool does not support deterministic recording and replay, in the experiments of this work we focus on single-threaded workloads, and plan to extend the analysis to multi-threaded workloads in future. Previous studies on recording and replay techniques for DBI [88, 108] (that are unfortunately still not implemented in the Valgrind framework at the time of writing) makes us confident that the approach is applicable to multi-threaded applications.

I/O operations. Similarly to thread scheduling, the timing and the contents of I/O operations can affect the execution flow and the sequence of *stores* of a program. Non-determinism due to I/O timing can be avoided if the effects of thread scheduling are avoided, either through recording and replay or by focusing on single-threaded applications: In the case of recording and replay, the deterministic thread scheduling makes the execution tolerant to variations in the timing of I/O operations; in the case of single-threaded applications, the execution is insensitive to I/O timing. Moreover, we avoid non-determinism of I/O contents by executing our target applications in a controlled experimental environment, in which the target is fed with the same I/O data (e.g., the same input files) at each execution.

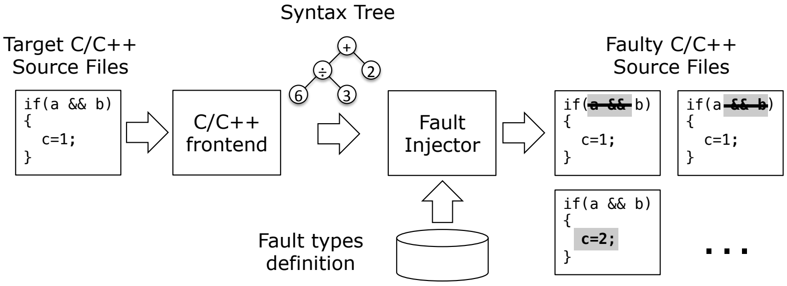
Random number generators. The use of (pseudo) random numbers in a program can lead to random values being written to memory and to variations of the execution flow. We avoid the effects of random numbers by wrapping random number generators, such as `rand_r`, and forcing them to return the same sequence of numbers at each execution.

4.3.2 Component fault injection

To inject software faults in library code, we use the approach and the automated tool (SAFE) described in [28,80]. The tool injects a set of *representative* fault types (4.1), which were defined on the basis of field data on *real software faults* found in deployed software systems, both commercial and open-source [25,36]. The SAFE tool injects these fault types by mutating the source code instead of the binary code, which assures a high degree of accuracy of fault injection experiments [29]. The tool automatically identifies *code locations* in which faults can be injected, and *code changes* for realistically emulating the fault types of 4.1. Each injected fault produces a distinct *faulty version* of the target library code (Figure 4.10), which replaces the original code.

Table 4.1: Fault types adopted in this study [36].

Type	**ODC**	**Description**
MFC	ALG	Missing function call
MIA	CHK	Missing IF construct around statements
MIEB	ALG	Missing IF construct plus statements plus ELSE before statements
MIFS	ALG	Missing IF construct plus statements
MLC	CHK	Missing AND / OR clause in expression used as branch condition
MLPA	ALG	Missing small and localized part of algorithm
MVAE	ASG	Missing variable assignment using an expression
MVAV	ASG	Missing variable assignment using a value
MVIV	ASG	Missing variable initialization using a value
WAEP	INT	Wrong arithmetic expression used in parameter of function call
WPFV	INT	Wrong variable used in parameter of function call
WVAV	ASG	Wrong value assigned to variable



```
if(a && b)
{
  c=1;
}
```

```
if(~~a &&~~ b)
{
  c=1;
}
```

```
if(a~~ && b~~)
{
  c=1;
}
```

```
if(a && b)
{
  c=2;
}
```

Figure 4.10: Software Fault Injection approach [28, 80].

The representativeness of injected faults is an important requirement for obtaining a realistic profile of interface errors generated by software faults. Compared to the mutation operators proposed in the literature for the C language, the considered fault types are more selective and only encompass faults most frequently found in the field (12 fault types against 71 mutation operators proposed in mutation testing studies [59]). This reflects the fact that mutation operators inject many kinds of faults that can occur before and during coding and are used to assess the thoroughness of test cases, while the fault types of 4.1 represent faults that tend to escape the whole development process (including testing), and are not designed for improving test suites but assessing fault tolerance properties. These fault types also provide several detailed rules ("constraints"), not shown for brevity, describing the *code context* in which fault types should be injected to be representative of field faults [36]. For instance, the removal of an `if` construct is injected in those `if` constructs that enclose at most 5 statements, since it is unlikely that an `if` construct is lacking for larger groups of statements. Moreover, the proportions of injected faults follow the distribution of fault types in the field [36]. Fault types are grouped in 4 classes, according to the Orthogonal Defect Classification (ODC) [24]: Assignment (ASG), Algorithm (ALG), Checking (CHK), and Interface (INT).

4.3.3 Results

In our fault injection experiments, faults led to the following outcomes:

- **Crash**: the experiment is terminated by the OS due to an exception (e.g., due to an invalid memory access).
- **Hang**: the experiment is stalled, i.e., it does not terminate within a given amount of time (much larger than the duration of a fault-free execution).
- **Wrong**: the experiment produces an incorrect output, i.e., different from the output in fault-free conditions.
- **Pass, corrupted**: the experiment produces a correct output, but interface errors are observed.
- **Pass, no corruption**: the experiment produces a correct output, and the fault did not cause interface errors.

4.2 provides the distributions of failure types for each target library. In many cases, the output of experiments was correct even in the presence of an injected fault. By analyzing interface parameter data exchanged at component interfaces, we found that, in the 61.8% of experiments, there were neither incorrect outputs nor corruptions at component interfaces: in these experiments, the fault was not activated (even if faulty code was covered by the workload), or there was no error propagation to component interfaces. This result demonstrates that efficiency can indeed be an issue for CM experiments as many injections do not produce effects on experiments [25]. In total, we obtained 1131 failures from fault injection

Table 4.2: Outcomes of experiments.

Target	**Outcome**				
	Crash	*Hang*	*Wrong*	*Pass, corrupted*	*Pass, no corruption*
Libxml2	70 (4.8%)	20 (1.4%)	233 (15.8%)	147 (10.0%)	1001 (68.0%)
Libbzip2	6 (1.3%)	0 (0.0%)	39 (8.4%)	83 (17.9%)	335 (72.4%)
SQLite	122 (11.9%)	16 (1.6%)	182 (17.8%)	213 (20.8%)	490 (47.9%)

experiments (38.2% of the total), on which we performed more detailed analyses on interface errors. This set of failures is much larger than previous studies [58, 77].

First, it is determined the extent of corruptions of interface parameters, in terms of number of bytes affected by faults. For the experiments that resulted in interface data corruptions, Figure 4.11 provides the empirical cumulative distribution of the number of corrupted bytes, for each target library. The number of corrupted bytes ranges from single (10^0) bytes to thousands of bytes for all three libraries and this number depends on the types of the data structures and library functions affected by the fault. An important result, which holds for all three targets, is that 50%-60% of faults affect much more than 8 bytes, which is the size of a memory word in our target system (i.e., the maximum size of addresses or data that CPU instructions operate on). Less than 40% of faults are limited to a memory word, while the median of the number of corrupted bytes ranges between 50 and 110 bytes. This is an important finding for the design of representative interface error models, since it indicates that the traditional ones based on the corruption of individual bits or bytes on heap, global, and stack areas [15,61], are not suitable for emulating interface errors produced by software faults. Figure 4.12 provides the distribution of the number of corrupted bytes, split by ODC fault types (see 4.1). The figure shows that only the 20-30% of Algorithm, Checking, and Interface faults affect at most 10 bytes. Only in the case of Assignment faults (e.g., a missing variable initialization), about 50% of faults affect at most 10 bytes, as in these cases the incorrect assignment affects individual fields of data structures returned to the main program. Nevertheless, a significant share of Assignment faults still corrupt large memory areas.

The analysis of return values pointed out that several types of values can be returned by library invocations affected by software faults. It is consider the return value of a faulty invocation as *incorrect* when it differs from the return value of the same invocation in the fault-free trace. 4.3 shows the distribution of incorrect return values, by classifying them into -1, 0 non-pointer data, $NULL$ pointers, and *wrong pointers/values*, i.e., return values different from fault-free executions that do not fall into any of the other classes. Moreover, we further distinguish between the case in which both wrong return values and memory corruptions occur after the same invocation, and the case in which a wrong value is returned without the corruption of memory areas. The distributions of return values depend on the data type returned by library functions, which vary across different targets. Most

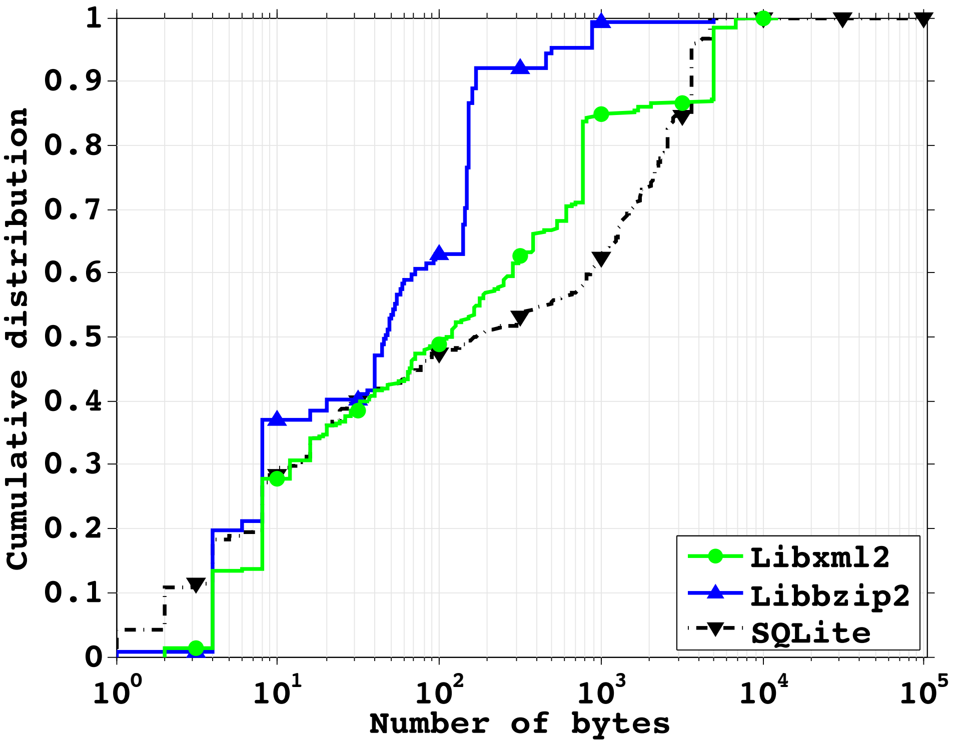


Figure 4.11: Cumulative distribution (per library) of the number of corrupted bytes of interface data.

importantly, we found that in most cases (75.2%) wrong return values are accompanied by memory corruptions. For instance, in the case of a library function that reads data from the disk and that behaves erroneously, both data returned through an input/output parameter (e.g., containing disk data) and the return value (e.g., representing the number of bytes read) become incorrect during the same invocation. This finding has significant implications for the injection of representative interface errors: existing tools that inject faults at library interfaces, such as FTS and LFI [48, 73], focus on the injection of wrong return values, but neglect the injection of memory corruptions. To achieve representativeness, wrong return

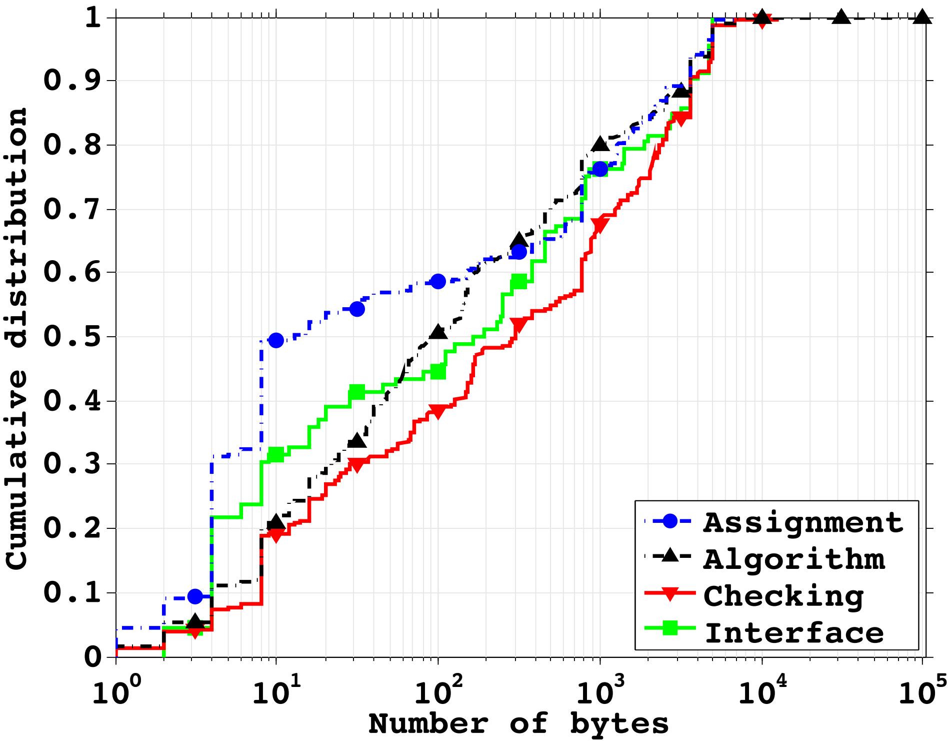


Figure 4.12: Cumulative distribution (per fault type) of the number of corrupted bytes of interface data.

values should be injected along with memory corruptions.

Furthermore, by comparing the number of failures with error codes (4.3) and the total number of experiments that lead to failures (4.2), we found that wrong return values only occur for a fraction of cases (40.9% for *Libxml2*, 75.6% for *Libbzip2*, 22.5% for *SQLite*). This indicates that "plausibility" checks that operate solely on return values are insufficient for detecting library failures, as several failures occur despite correct return values from library functions.

Table 4.3: Distributions of return values in fault injection experiments.

(a) Return values, without memory corruption.

Target	-1	*NULL ptr*	0	*Wrong ptr*	*Wrong value*
Libxml2	5 (3.8%)	0 (0%)	4 (3.0%)	0 (0%)	50 (37.9%)
Libbzip2	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
SQLite	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)

(b) Return value, with memory corruption.

Target	-1	*NULL ptr*	0	*Wrong ptr*	*Wrong value*
Libxml2	32 (24.2%)	33 (25.0%)	1 (0.8%)	5 (3.8%)	2 (1.5%)
Libbzip2	0 (0%)	4 (11.8%)	0 (0%)	28 (82.3%)	2 (5.9%)
SQLite	0 (0%)	2 (2.8%)	0 (0%)	1 (1.4%)	69 (95.8%)

As pointed out in the previous analysis of memory corruptions, the amount and location of corrupted data varies with the target library, and in particular with the type of interface parameters and of library functions. To support the tuning of representative IEI experiments for a specific target library, we investigated a heuristic rule for selecting which memory areas of interface parameter data to corrupt. In particular, the heuristic should allow to identify which memory addresses have the highest *corruption rate*, that is, memory addresses most likely corrupted by software faults, in order to focus error injections on them.

To identify such a heuristic, we analyzed memory accesses of the target libraries during fault-free executions, which can be easily obtained through a DBI analysis. We obtained the corruption rate of each memory address by computing the percentage of our fault injection experiments that led to the corruption of that memory address. We found that *the corruption rate of a memory address is strongly correlated with the number of accesses on that*

Table 4.4: Correlation between corruption rate and number of accesses.

●textbfTarget	**Spearman's ρ**	**p-value**
Libxml2	0.953	~ 0
Libbzip2	0.572	~ 0
SQLite	0.883	~ 0

memory address in fault-free executions. Figure 4.13 shows the growth of the corruption rate with the number of accesses in fault-free conditions for *Libxml2*. The quantitative analysis of correlation (4.4), using the Spearman correlation coefficient for ordinal data (which can be applied even when the association between elements is non-linear) and a statistical hypothesis test (with the null hypothesis that there is a zero correlation) [98], confirm this observation:

We found that the corruption rate and the number of accesses have a statistically significant correlation (the null hypothesis can be rejected at any reasonable type I error level, as the *p-value* of the test is lower than the smallest representable float number on our machine). From this result, we conclude that a heuristic rule for obtaining a realistic error model is to corrupt those memory addresses that are most often accessed in fault-free executions.

4.4 Summary

In this chapter, it is proposed an approach for analyzing how software faults in library code manifest as interface errors. We analyzed interface errors in three real-world libraries, obtaining guidelines for representative error injection experiments. In future work, we aim

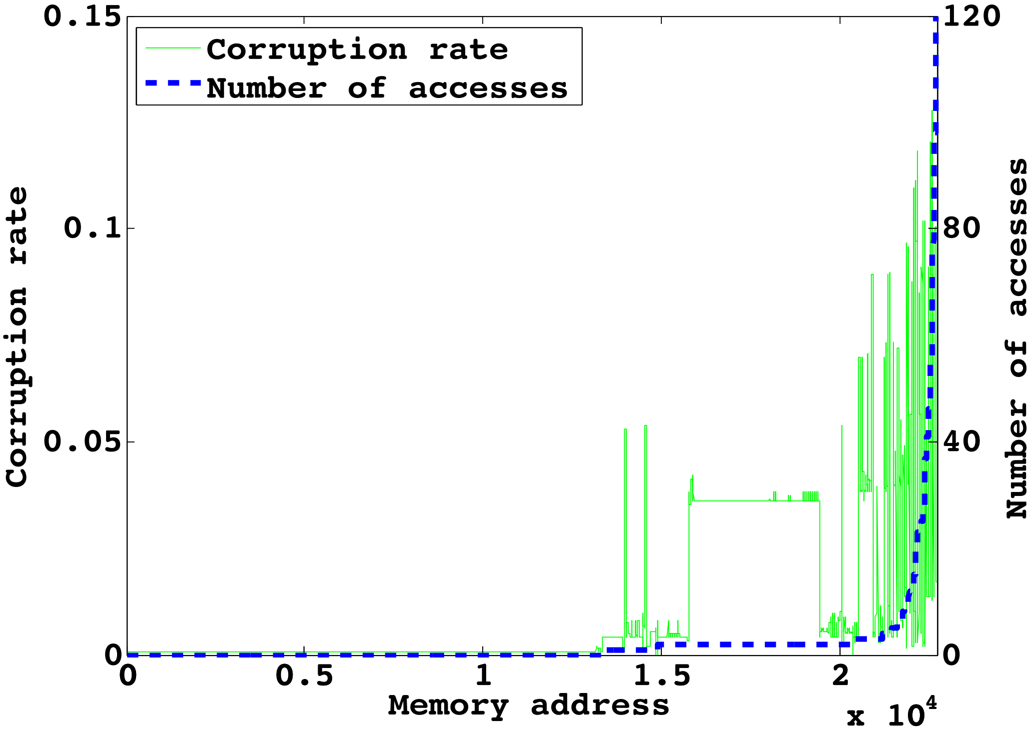


Figure 4.13: Byte corruption rate and number of accesses for *Libxml2*.

at applying these findings in IEI experiments, and investigating whether they can improve the representativeness of results.

We identified the following threats to validity: (i) the effects of non-determinism, that can lead to variations of memory *stores* that are not actually due to faults, and thus could mislead our the analysis of memory corruptions; (ii) the use of fault injection in components' code, in place of real software faults, to generate and analyze interface errors; and (iii) the selection of the target libraries.

As for non-determinism, we carefully designed our approach to factor out its effects from execution traces, and validated its ability to avoid non-deterministic interferences by verifying that execution traces are exactly reproducible when no fault is injected. The limitation of our analysis is that, in this initial phase of our research, we chose to first focus on single-threaded executions, before extending our approach to multi-threaded executions using recording-and-replay techniques as discussed in Section 4.3.1. The use of real faults for obtaining real interface errors is unfortunately hampered by the shortage of faults to analyze for specific library versions and configurations, especially in the case of very mature and highly reliable software such as SQLite. We therefore adopted fault injection, which allows to perform a high number of experiments and, at the same time, is able to generate representative errors, as demonstrated by several empirical studies on the use of code mutation for software engineering experimentation [9, 30, 31]. We are thus confident that the validity of our findings is not significantly affected by the use of fault injection. Conclusions are based on experimental results from three libraries, and may not generalize to all types of libraries. As the chosen libraries are widely used and functionally diverse, we believe that the obtained results are representative for a larger set of libraries. Apart from whether a generalization is valid, we demonstrate that libraries with the discussed error manifestations exist and that existing error models do not match these manifestations. Moreover, we provide an approach that is suitable to analogously assess any library of interest.

Chapter 5

Software-Implemented Fault Injection in the Multicore Era

5.1 Introduction

The trend of integrating OTS components in safety-critical systems involves also hardware components because they allow us to develop more innovative and performing solutions in a cost-efficient way. Nevertheless, the adoption of sophisticated OTS hardware devices such as multicore processors increases system complexity and raises novel dependability challenges: the number of cores, concurrency issues, shared resources and interconnections among cores make it hard to develop and validate software deployed on the top of multicore processors.

Compared to single-processors, multicores assure higher performance by integrating more cores on one die that, running in parallel and sharing many resources (e.g. memory, caches, registers). The complexity of the architecture increases the probability of hardware errors because of the great number of transistors on the same chip. For this reason, designers and developers of modern processors have enhanced hardware implemented mechanisms in order to make them more reliable against errors. In fact, hardware mechanisms for memory, cache and registers protection are significantly improved: error detection mechanisms are able to signal to the upper software layer detected hardware errors. In multicore processors, these mechanisms are even more complicated because they have to take into account the presence of more cores. So that, the complexity is shifted to the software that has to correctly interpret the signaled hardware errors and to implement adequate software recovery mechanisms to cope with them. This opens new scenarios to fault injection techniques employed to validate fault tolerant mechanisms.

Traditionally, software-implemented fault injection were successfully adopted for dependability assessment. Most of these techniques emulate the effects of hardware errors through software using fault model based on bit-flip and stack-at. Although these techniques can be applied also when system integrates multicore processors, fault injection campaigns could be very expensive and infeasible due to the great number of added resources. In fact, the replication of cores means also replicated resources such as per-core and shared registers and caches. Moreover, the injection of single or multiple bit-flip errors can be masked to the software by the advanced hardware protection mechanisms that, in many cases are able to correct it. This makes fault injection techniques based on bit-flipping and stack-at models not effective. In addition, these fault models do not encompass same resources proper of multicore (that were not a concern for single-core processors) such as the communication links and added cache levels shared among cores: this affect the representativeness of fault models when fault injection are used for dependability assessment of systems running on the top of multicore.

This chapter presents a novel error injection framework for supporting the validation of fault tolerant mechanisms of the existing operating systems running on the top of OTS multicore processor by exploiting the error reporting mechanisms of modern processors. The approach was also extended to virtualized systems aiming at evaluating the error handling mechanisms of existing hypervisor deployed on multicore.

5.2 Multicore in safety-critical systems

Recently, embedded and safety-critical market looks with interest to modern multicore processors due to the success already obtained in many systems such as personal computer, server, tablet and, smarthphone. It is recognized that multicore processors brings many advantages in terms of performance and power consumption.

Nevertheless, it is important to underline that the integration of OTS multicore processors in safety-critical systems is not driven by the need of solving problems using simultaneously multiple computational units. Indeed, this would require several and difficult changes in software components: programs should exploit thread level parallelisms introducing more potential software bugs (e.g. race conditions and deadlocks) and temporal non-determinism of the execution. Moreover, it is still an open issue how to schedule threads (including when and how often they are scheduled and how they are distributed among cores). Although, in literature there are studies that investigate the use of multicore in safety-critical systems such as *avionic* [65], [7], *automotive* [12], [82] and *medical* [119], we are still far from using software that fully leverages concurrency and thread parallelism [41]

Instead, multicore processors in safety-critical systems seems to be a good solution to reduce hardware units. The integration of multiple functionalities on the same device lead to a significant reduction of hardware units improving the reliability of the entire system: components on the same board means also less number of link connections that are recognized to be a significant cause of system failure as discussed in [106]. Having more processing units can accelerate the migration process from traditional federal architecture, where each component is independent and executes a single task, to an integrated architecture where tasks are assigned to different core on the same device. Integrated Modular Avionics (IMA) [100] in the aerospace domain and AUTOSAR [1], [82] in the automotive domain are examples of this trend. Traditionally, in modern cars, there are several Electronic Control Units (ECUs) connected by communication links, each ECU is responsible of executing a single task . For instance, different ECUs are responsible for functions like gear control, engine control, and adaptive cruise control. Recently, moving to an integrated architecture, more ECUs are on the same hardware device. This requires additional functionalities to be implemented in software and, in particular, in the operating systems. For instance, the last version of AUTOSAR [1], standard de-facto for automotive, added support for multicore architecture. The operating system includes some capabilities such as static assignment to a core, inter-core and intra-core coordination to access shared resources. However, it does not allow dynamic tasks assignment to a core. AUTOSAR OS protection mechanisms to support software components isolation and scheduling issues are described in [82].

Hardware consolidation by means of multicore requires mechanisms that assure the isolation between cores, i.e. applications running on different cores do not have to interfere each other. A valuable solution is provided by the adoption of virtualization in conjunction with virtualization [33]. By means of the virtualization, multiple operating systems and applications run on the same physical board in separate partitions, named *virtual machines* communicating with the hardware through a software layer, the hypervisor. Besides hardware consolidation, virtualization also allows us: (i) software consolidation, i.e. different operating systems (e.g. real-time and not real time) and applications run on the same device [52], [68]; (ii) to develop mixed-criticality systems, i.e. safety and non-safety applications are isolated and they can run simultaneously on the same platform; (iii) to execute legacy systems; (iv) to gradually migrates to COTS technologies and to new hardware devices [104]; (v) to implement fault-tolerant mechanisms based on replication and/or diversity approaches [43]; (vi) to reduce certification costs by isolating previously certified software; (vii) to improve system robustness and reliability by protecting and isolating operating environments from each other.

For these reasons, virtualization-based solutions were also investigate by safety-critical industries such as aerospace [93], [45] and automotive [81]. Moreover, commercial solutions were proposed by WindRiver, [6] in collaboration with Intel [6], [117] towards a certified hypervisor.

5.3 Background and Related Work

Once an OTS hardware component is acquired and integrated, testing activities are required in order to evaluate system behaviour in presence of hardware faults. Several fault injection techniques were proposed in order to inject **hardware faults**. Hardware-based fault injection techniques insert into the system real hardware errors by means of special-purpose and architecture-dependent equipment or by interfering with the physical unit (e.g., by lowering the device voltage, increasing the temperature, radiations introducing electromagnetic interferences) [50]. This approach has the advantage of reproducing real hardware faults, but it is costly and risky to implement. Moreover, it makes it hard the observation of the effect of the faults in the processor because of the interferences caused by the injectors. For these reasons, **software-implemented** fault injection (SWIFI) techniques have gained popularity. SWIFI consists of reproducing via software the effects of hardware errors. The injection can be performed at compile time inserting the effects of hardware errors in the target code or at run time using time-out, exceptions or code insertion to trigger the fault injection. Several tools implementing SWIFI technique were proposed in literature above:

- FIAT [15] corrupts the data area of the binary according to three fault models, namely, zero-a-byte faults, set-a-byte faults, and two-bit compensating faults. The zero- a-byte and set- a-byte faults zeros or sets eight bits of a 32 bit word, two-bit compensating faults flip two bits. Experiments did not consider the injection of a single bit because the hardware was equipped with parity check.

- FERRARI [61] could inject permanent and transient faults as well as control flow errors, bus errors, memory errors, and processor control line errors into systems based on SPARC processors from Sun Microsystems. FERRARI uses software traps to inject faults and has five fault models: XORing a bit, resetting a bit, setting a bit, setting a byte and resetting a byte.

- FINE [64] emulates hardware and software faults 2 on the kernel of Sun OS 4.1.2. FINE can inject transient and permanent hardware faults in the CPU, bus and memory (text and data area) by flipping a bit.

- DEFINE [63] is the evolution of FINE for distributed systems. Basically, DEFINE injects faults in a single node as FINE does, in addition observe if and how they affect other nodes in the system.

- DOCTOR [51] can inject communication faults as well as traditional hardware faults such as memory and CPU faults into HARTS distributed system. The faults can be intermittent, permanent and transient. Fault can be injected as a single bit, two-bit (compensating), whole byte, or burst (of multiple bytes). Communication faults in DOCTOR can cause messages to be lost, altered, duplicated, or delayed.

- FTAPE [111] performs injection on TANDEM system and sup- ports single/multiple bit-flip and zero/set faults in CPU registers (e.g., stack pointer, program counter) as well as in memory. FTAPE also includes I/O faults, that is, SCSI and disk faults.

- Xception [22] takes benefit of the exception available on the microprocessor, i.e., execute a run time injection. The fault models includes flip stuck-at-zero, stuck at-one, and bit flip.

- EXFI [17] exploits the Trace Exception Mode available in most low cost micropocessor. EXFI can inject single bit-flip transient fault into memory data and registers. A notable feature of this tool is a set of fault collapsing rules which reduces the number of faults to inject without decreasing the accuracy of the results.

- MAFALDA [94] corrupts pseudo random selected byte in the code segment and data segment of a Microkernel OS. MAFALDA can flip one or more bits for a temporarily, i.e., emulates a transient error.

- Skarin [99] extends the previous version of Goofi [8] to inject multiple bit flip and bit flip into CPU registers and memory. Goofi-2 sup- ports both pre-injection and run-time injection. A notable future of Goofi-2 is the optimization of the fault-space by utilizing assembly-level knowledge of the target system in order to place single bit-flips in registers and memory locations only immediately before these are read by the executed instructions.

However, the use of fault injection techniques for the assessment of **multicore systems** is still recent. Appropriate fault models encompassing faults that were not a concern in single-core architectures (e.g., adopting SWIFI technique, the execution of additional software for the injection could affect the scheduling of the system tasks impacting real-time requirements) are required to guarantee effective and low cost fault injection campaigns. Challenges in tolerating faults in parallel execution on multicore systems are discussed in [79]. In [97], mSWAT is presented. It is a detection and diagnosis technique for permanent and transient hardware faults in multicore architectures running multithreaded software. The authors adopt fault injection by simulation in order to validate the detection mechanisms. However, assuming that at most one core is faulty, the fault model encompasses only in-core faults and not faults that can occur in I/O controller, memory sub-system, etc. In both [40] and [70] a simulation-based fault injection analysis for multicore is presented. In [57] the use of NFTAPE tool for the evaluation of operating system behavior running on multicore processor is proposed. In [95], the authors describe a method for predicting failures based on the monitoring of the execution units in a Quad-core Intel processor.

5.3.1 Software-implemented Error Injection for Multicore

The proposed approach of error injection leverages the notion of machine check errors. Machine check errors (MCE) indicate the occurrence of problems affecting hardware units of the processor. Modern processors usually notify MCEs by means of an error-reporting architecture (exemplified in Figure 5.1) composed by a set of *global* and *per-core* registers. The idea underlying our proposal is emulating the occurrence of MCEs by **writing into the registers** of the error-reporting architecture rather interfering with the device, such as in the mentioned hardware fault injection approaches. The knowledge about MCEs and error codes reported by the processor during the execution is inferred from the documentation provided by the manufacturer of the processor [3]. Modifying the registers of the error-reporting architecture allows implementing a low-cost and controllable fault injection framework.

Figure 5.1 shows the functional components of the framework implementing the injection approach. The system under test is composed by the multicore processor and a **target workload**. The latter could be an operating system, software for embedded systems, or a virtualization-based solution, and represents the software whose robustness is assessed under the occurrence of MCEs. The role of the remaining components depicted in Figure 5.1 is described in the following along with relevant design challenges:

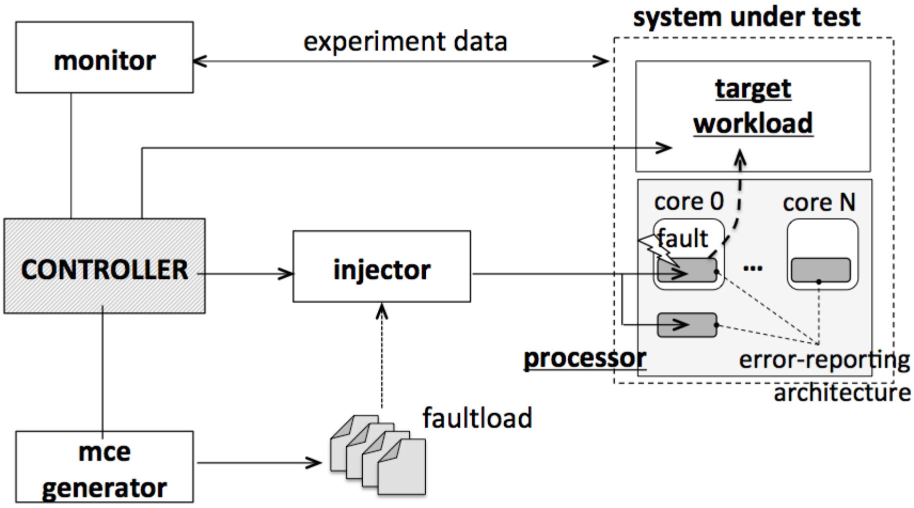


Figure 5.1: Proposed Error Injection Framework

- **MCE generator** is the entity that automates the generation of the *faultload*, i.e., the set of MCEs that will be injected in the target system during the campaign. It should be noted that the number of combinations representing all the possible error codes reported by the processor is extremely large. The faultload generation should be optimized with criteria aiming to narrow down the number of experiments. For examples, experiments might focus on a given hardware unit or specific border values assigned to the error-reporting registers of the processor.

- **Injector** is the component responsible for injecting MCEs into the error-reporting architecture of the target processor, as shown in Figure 5.1. The injector should not distort the actual behavior of the system under test. For this reason, if injection is accomplished via a software module, isolation between the injector and the injection target can be achieved by running them on different cores. A better solution is represented by the use of specialized hardware supports, such as a debugger; however, this might not always be available to analysts and requires additional costs. Even of more relevance, injector must address spatial and timing features of the experiment.

- **Monitor** is responsible for collecting data concerning the fault-injection outcomes. Data might include notifications reported in the system log, output produced by the target workload, or state variables. Monitor should cope with data loss caused by experiments leading to critical system failures, such as reboot or panic. Again, monitoring and data collection features should not impact the behavior of the target system.

- **Controller** is the entity responsible for iterating fault injection experiments and coordinating the described components. For each experiment it activates/deactivates the injector module, and stores monitoring data. Moreover, controller should ensure that the workload is actually running at the time injection is performed. To this objective, controller might leverage operating system support (e.g., Linux OS get/set CPU affinity mask) to cope with processes scheduling issues.

5.3.2 Case Study

The multicore processor targeted by the study is the **Intel Core i7** 2670QM [3]. Figure 5.2 shows a simplified block diagram of the architecture. It is a distributed shared memory

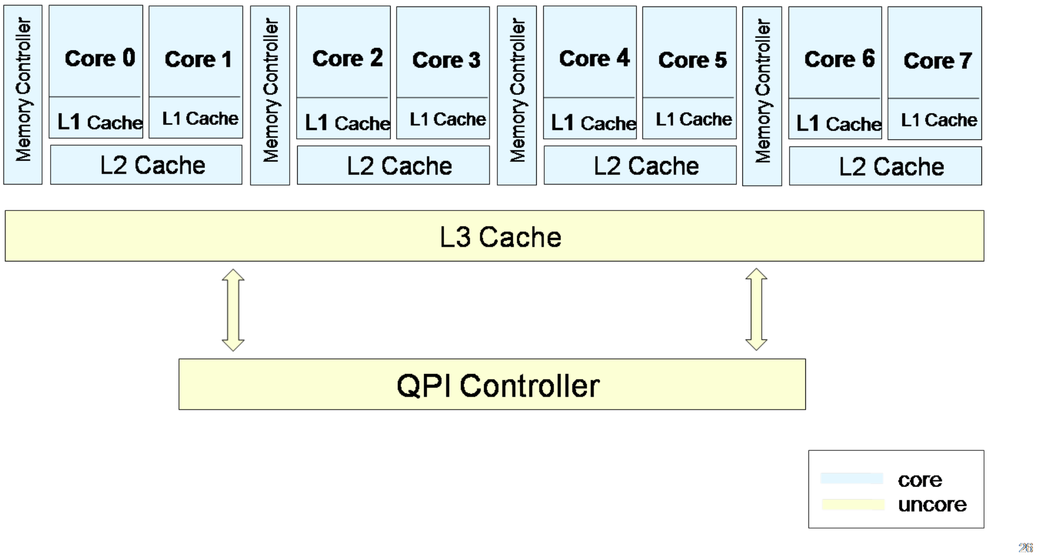


Figure 5.2: Intel Core i7 Architectural Block Diagram

system consisting of 4 physical cores integrated on the same chip. Cores are connected by a point-to-point and high-speed communication link (Quick Path Interconnect). Each core appears to software as two logical cores by means of the Hyper Threading Technology (e.g. Intel's implementation of Simultaneous Multi-Threading). Moreover, the processor introduces several new features (e.g., integrated memory controller for each core, a memory hierarchy with 3 caches levels) that assure high performance and power efficiency.

The processor provides a sophisticated error-reporting architecture called **Machine Check Architecture** (MCA). The MCA is composed by a set of registers (Machine Specific Register - MSR) for reporting errors detected by hardware components, such as memory, caches, and buses. As shown in the Figure 5.3, the MCA consists of 9 banks of registers

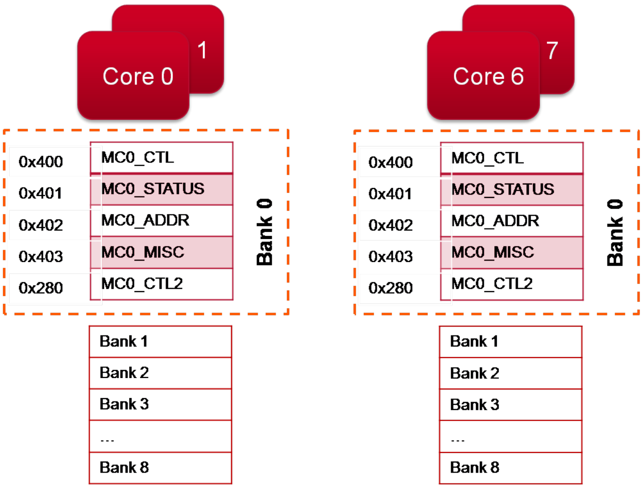


Figure 5.3: Machine Check Architecture

replicated for **each core** and associated to specific hardware units. Each bank is composed by 5 registers for reporting hardware errors: two *control* registers (`MCi_CTL` and `MCi_CTL2`), a *status* register (`MCi_STATUS`), an *address* register (`MCi_ADDR`), and a *miscellaneous* error information register (`MCi_MISC`). By means of the bank of registers, the MCA notifies different category of errors:

- **uncorrected errors** (UC): errors not corrected by the processor;
- **uncorrected recoverable errors** (UCR): errors not corrected by the processor and for which system software can attempt recovery actions. In particular, the recovery actions can be *required* (SRAR), *optional* (SRAO) or *not required* (UCNA).
- **corrected errors** (CE): errors corrected by the processor without impacting running processes.

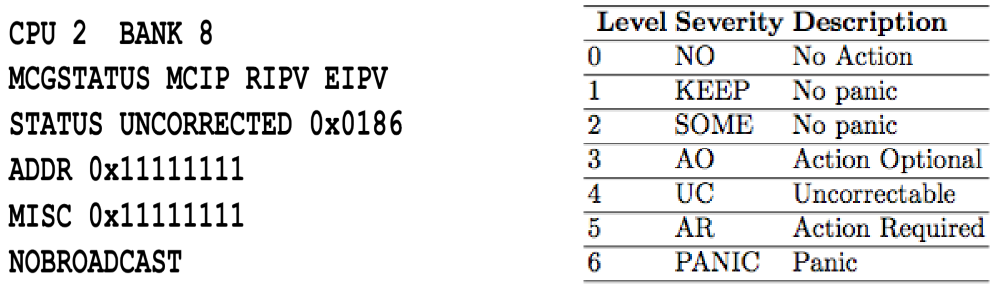
Additional information about the type of errors affecting hardware units of the processor is available in the first 16 bits of the status register (again, `MCi_STATUS`). Details about the error codes adopted in the study are reported in the Table 5.1, which has been taken from the processor documentation. This information supported the definition of realistic fault model for the target processor. Moreover, 3 **global registers** are provided by the processor: `MCG_CAP`, `MCG_STATUS` and `MCG_CTL` registers. In the context of our work, we considered the `MCG_CAP` register, which gives information about the capabilities of the MCA available in the processor (e.g., bit `SER_P` of such register indicates if the processor supports recovery actions) and `MCG_STATUS` register, which reports the status of the processor at the time a MCE occurs.

A preliminary implementation of the framework described in Section 5.3.1 has been

Type	**Format***
Generic Cache hierarchy	0001 0000 0000 11LL
TLB	0001 0000 0001 TTLL
Memory controller	0001 1MMM 11LL CCCC
Cache hierarchy	0001 0001 RRRR TTLL
Bus and interconnections	0001 1PPT RRRR IILL

**TT* - Type of transaction
LL - Level in the memory hierarchy
RRRR - Type of action associated with the error
MMM and CCCC - Memory transaction type and Channel
PP and T - Partecipation and Timeout
II - Memory or I/O

Table 5.1: Status Register [15:0]



```
CPU 2  BANK 8
MCGSTATUS MCIP RIPV EIPV
STATUS UNCORRECTED 0x0186
ADDR 0x11111111
MISC 0x11111111
NOBROADCAST
```

(a) Machine Check Error

Level	Severity	Description
0	NO	No Action
1	KEEP	No panic
2	SOME	No panic
3	AO	Action Optional
4	UC	Uncorrectable
5	AR	Action Required
6	PANIC	Panic

(b) Linux Severity Levels

Figure 5.4: MCE Description File and Severity Levels

developed under the Linux OS. Implementation has been used to conduct explorative fault-injection campaigns to validate the proposed approach in a real testbed adopting the Intel i7 processor.

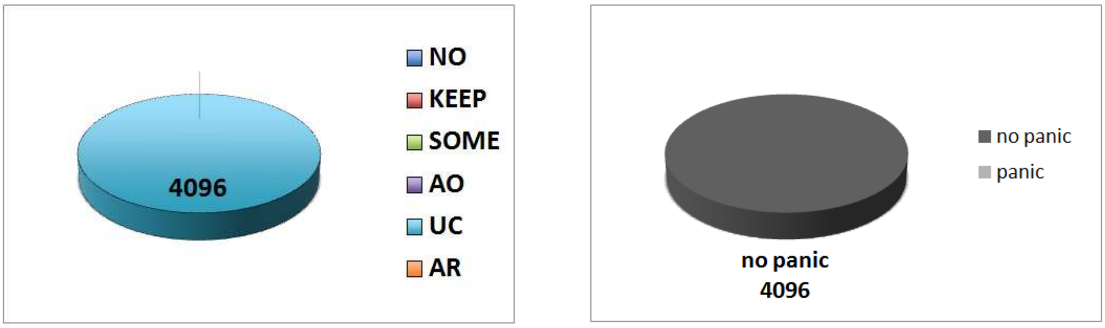
In the proposed implementation the **injector** consists of `mce-inject` [66], which is a well-known tool in Intel/Linux community. Each MCE is represented by a textual description providing information about the location (i.e., cpu and bank number) where the MCE will be injected and values assigned to `MCG_STATUS`, `MCi_STATUS`, `MCi_ADDR`, and `MCi_MISC` registers of the MCA architecture. Figure 5.4a provides an example of MCE to be injected in the *bank 8* of the *cpu 2*. It emulates an *uncorrected error* affecting data of L2 cache during the snoop protocol by means of the error code *0x0186* that will be written in the status register. A `bash` script has been implemented to automatically generate the *faultload*, i.e., the set of MCEs that are injected during a campaign. Given the textual description, *mce-inject* sets the values of the registers of the MCA by means of a specific kernel module of the Linux OS.

The **workload** is represented by the Linux OS (kernel version 3.1.10) running on the top of the Intel i7 processor. Preliminary experiments aim to explore the Linux error-handling capabilities initiated by the `do_machine_check` procedure. This is the OS exception handler that is actually triggered when a real machine check occurs (interrupt 18 in the case of Intel processors). For each MCE-injection experiment, the **monitor** component collects the error *severity* determined by the kernel as a result of the injected MCE and the *recovery action* triggered by the kernel based on the severity level. Values assumed by the severity parameter under the Linux OS are reported in Figure 5.4b.

5.3.3 Campaign #1

The faultload of the first campaign consists of 4,096 MCEs. It emulates cache, memory controller, and TLB errors by changing the bits of the status registers according to the codes reported in Table 5.1. The set of emulated errors contains uncorrected recoverable errors (UCR), uncorrected recoverable errors with action required (SRAR), optional (SRAO) or not required (UCNA). Moreover, the same faultload has been injected into two different scenarios, i.e., (i) with the processor not supporting recovery actions (i.e., the bit `SER_P` is clean); (ii) with processor supporting recovery by software (i.e., the bit `SER_P` is set).

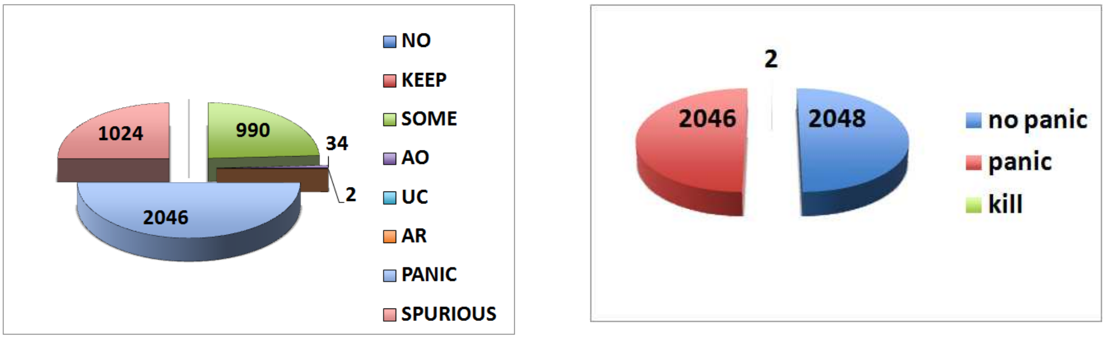
Figure5.5a and Figure5.5b show the MCEs severities and related recovery actions provided by the Linux OS, when the **bit SER_P is clean**. All the errors have classified as uncorrectable; however, this set of experiments did not cause the triggering of any specific



(a) Severity

(b) Recovery Actions

Figure 5.5: SER_P=0: Recovery Actions Not Supported by the Processor



(a) Severity

(b) Recovery Actions

Figure 5.6: SER_P=1: Recovery Actions Supported by the Processor

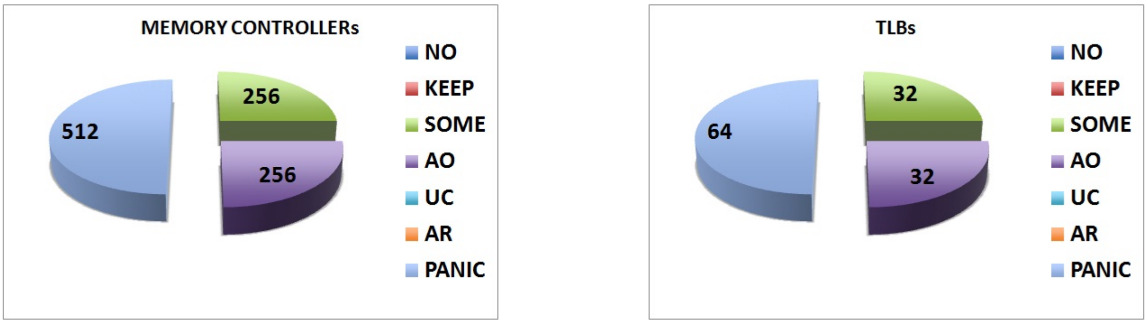
a recovery action.

The same set of 4,096 errors has been emulated with the **bit SER_P set**. Experiments made it possible to highlight a rather different behavior of the error handling mechanism indicated by Figure 5.6a and Figure 5.6b, respectively. Figure 5.6a shows the severity levels. The 50% of the injected MCEs is classified as PANIC: as a result, the injection of this subset of errors actually caused the panic of the machine, such as reported in Figure 5.6b. Errors causing SOME and AO severities represent total 24% and the 0.8%, respectively. These errors did not trigger any specific action of the handler. Only 2 error codes affecting the cache unit, i.e., `0x0134` - *data load error* and `0x0150` - *instruction fetch error* were actually recognized by the handler, i.e., AR severity, and caused the 2 process kills shown in Figure 5.6b.

More important, experiments revealed a possible bug in the code that determines the error severity (Figure5.6a). Around 25% of experiments caused a **spurious** severity value, i.e., a numeric value that is not a severity level according to Figure 5.4b. Spurious values prevent to correctly determine the severity of errors leading to unexpected behaviors and were attributed by the handler to the NO PANIC category Figure 5.6b. We also observed that the kernel does not strongly differentiate among errors affecting different hardware units. Figure 5.7a and Figure 5.7b report the distribution of the severities observed for memory controller and TLB errors with the bit SER_P being set. In both cases, the 50% of the errors causes a system panic regardless of the nature of the injected errors. Again, total 25% of errors caused a spurious severity.

5.3.4 Campaign #2

Because of the inability of the handler to differentiate among error codes, a further campaign has been performed to explore its recovery behavior. In this campaign, rather than exhaustively trying different error codes, we used different combinations of the diagnostic



(a) Memory Controller

(b) TLB

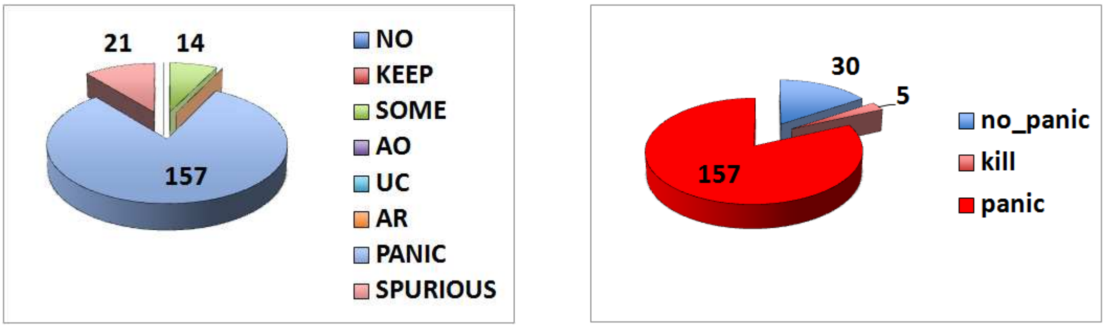
Figure 5.7: Severity and Recovery Actions Grouped by Error Categories

information provided by the MCA along with the error notification (such as, the error is recoverable or not, an action is required to recover from the error, the error corrupted or not the processor state, etc).

The campaign encompassed 192 MCEs injected when the bit SER_P is set, i.e., recovery actions are supported. Results reported in Figure 5.8a and 5.8b confirm that the target handler mainly provides coarse-grained recovery actions, i.e., system panic (82%) and process kill (3%). Again, the handler was not able to correctly manage around 11% of the errors due to the presence of spurious severity values, possibly causing an improper recovery action.

5.4 Emulating Hardware Errors in Virtualized Systems

An extension of the method proposed in 5.3.1 was implemented in order to emulate hardware errors in virtualized systems. The tool emulates the occurrence of hardware errors, i.e MCEs, by injecting the corresponding error codes in the registers for error reporting implemented in the hypervisor. The main goal is to evaluate the MCE handler mechanisms of the hypervisor



(a) Severity

(b) Recovery Actions

Figure 5.8: Severity and Recovery Actions for Campaign #2

and its ability of masking hardware errors to the guest operating system. The general schema is depicted in Figure5.9.

The hypervisor is the software layer that performs system virtualization, i.e. it is able to logically separate partitions, named virtual machines, from the hardware they run on. Among the virtualized components that the hypervisor provides to its virtual machine,

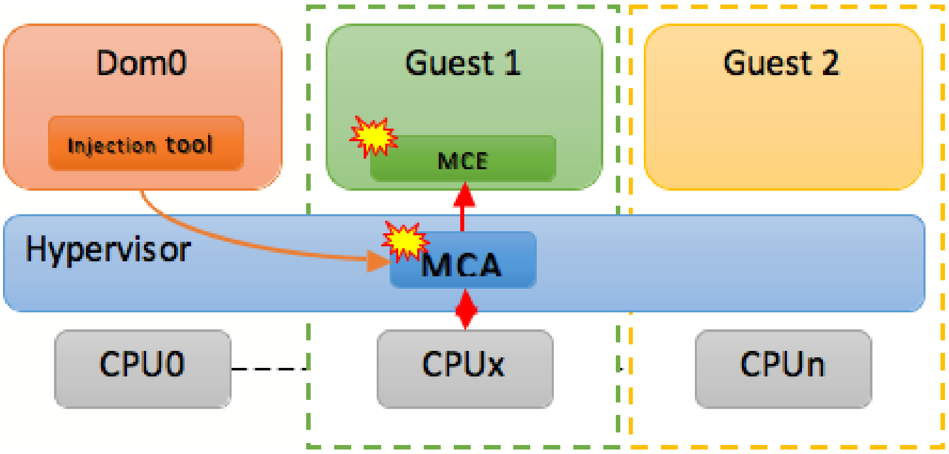


Figure 5.9: Injection Framework for Virtualized Systems

there are also the registers of the MCA. This makes possible the implementation of the error injection tool in order to evaluate its error handler mechanisms and how it protects virtual machines when hardware errors occur. Examples of hypervisor are Xen [91] and VMWare ESX [5]

5.4.1 Case study

A preliminary implementation of the framework has been developed to inject MCEs in Xen hypervisor installed on a Intel Core i7 processor. The proposed tool is able to inject hardware errors in the virtual representation of the MCA by accessing to it through a special high-privilege Xen administrative domain, known as *Dom0*.

The architecture of the framework for injecting MCEs into the MCA of implemented by Xen is depicted in Figure 5.10.

The injection tool is an extension of the one described in 5.3.1 and is composed by: (i) the MCE generator that, based on same specific criteria, generates the faultload, i.e. the list of hardware errors to be injected; (ii) the injector is the tool responsible for emulating MCEs in the registers of the MCA implemented by Xen; (iii) the coordinator is the arbiter of fault injection campaigns and it collects the results by analysing logs produced by both Xen and the guest operating system; (v) the guest operating system is the workload. To assure the isolation of the workload from the injection tool in order to avoid possible interferences that may compromise the experimental results, they are executed on dedicated cores.

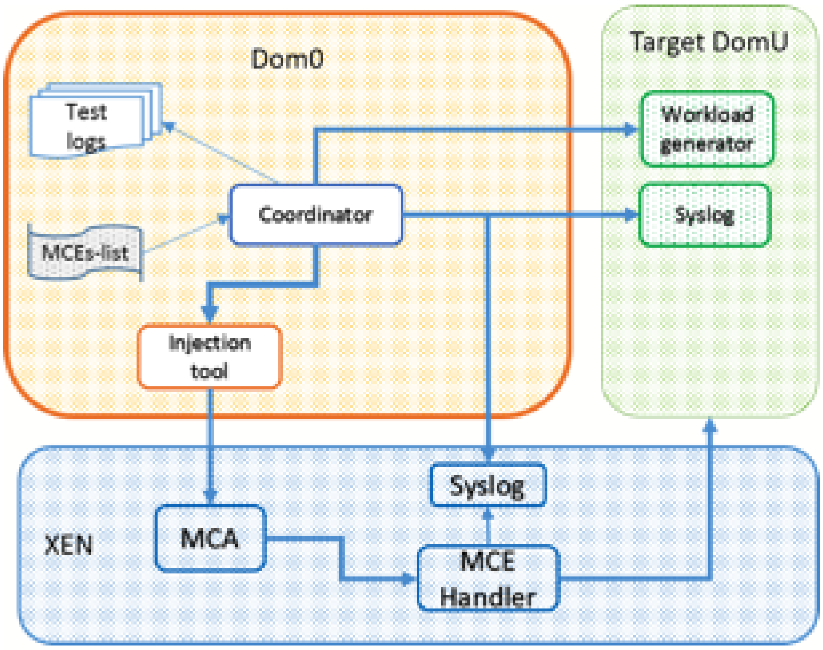


Figure 5.10: Architecture Framework for Xen

Option	Description
s [MSR value]	MSG_STATUS value
S [MSR value]	MCi_STATUS value
p [MSR value]	MCi_ADDRESS value
M [MSR value]	MCi_MISCV value
B [Bank number]	Bank number
c [CPU value]	CPU target
d [domain]	Target domain
g The error is sent to the guess	If presents, it disables XEN handling mechanism.

Table 5.2: MCEs injector input

Injection tool exploits the HYPERVISOR_mca hypercall for writing specific values in the MSR registers. The injection is performed by specifying the input parameters listed in Table5.2

An example of usage is:

```
./injection_tool -s 0x5 -S 0xbd80000000017a -c 4 -B 8 -d 3 -p 0x300 -M 0x1 -g
```

It represents the MCE error described in Table 5.3

MCG_STATUS	0x5
MCi_STATUS	0xbd80000000017a. It corresponds to SRAO L3 explicit writeback
MCi_ADDR	0x300 0x300 0x300
MCi_MISC	0x1
Bank	8
Target CPU	4
Target DOM	4
XEN handler	disabled

Table 5.3: MCE example

MCE generator automatically generates the faultload, i.e. the list of MCEs that will be injected. However, the combinations representing all the possible error codes reported by the processor is extremely large. It means that an exhaustive fault injection campaigns consists in a great number of experiments.

$$2^{nrbitMCG\_STATUS} x 2^{nrbitMCi\_STATUS} x nrbanks = 2^3 x 2^{64} x 9 \tag{5.1}$$

To lower the number experiments, it is possible to create campaigns based in specific criteria. For instance, it is possible to generate all the valid combinations for a fixed type of errors, or for a fixed CPU/bank.

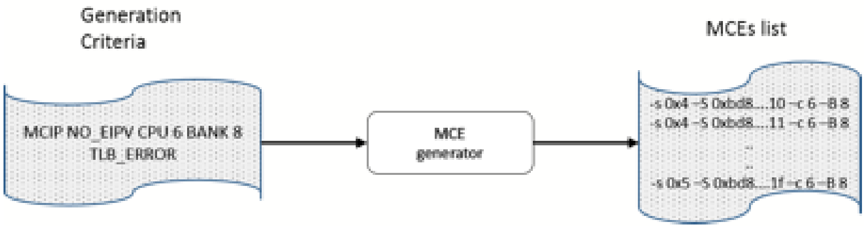

Generation Criteria
MCIP NO_EIPV CPU 6 BANK 8 TLB_ERROR
MCE generator
MCEs list
-s 0x4 –S 0xbd8....10 –c 6 –B 8
-s 0x4 –S 0xbd8....11 –c 6 –B 8
..
..
-s 0x5 –S 0xbd8....1f –c 6 –B 8

Figure 5.11: MCE Generator

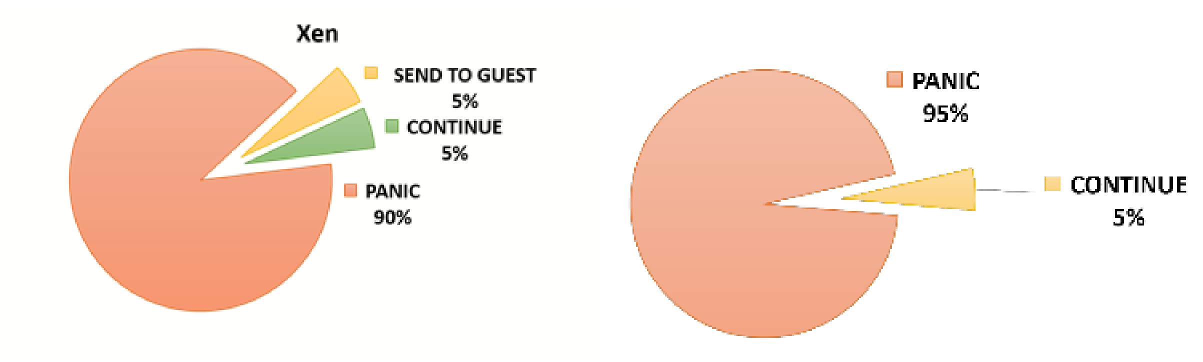
5.4.2 Campaign 1

The faultload of the first campaign consists of 49,502 MCEs. It encompasses cache, memory controller, and TLB errors by changing the bits of the status registers according to the codes reported in Table 5.1. The set of emulated errors contains uncorrected recoverable errors (UCR), uncorrected recoverable errors with action required (SRAR), optional (SRAO) or not required (UCNA). Moreover, the same faultload has been injected by targeting the CPU 6 and bank 8 where it is created a domU on which a Linux OS is running. Two different experiments are performed: (i) the errors are injected in the hypervisor context, i.e. the hypervisor is completely responsible of taking appropriate recovery actions; (ii) the errors are injected in guest context, i.e. the domain is responsible of sending occurred errors to the guest.

Figure 5.12a shows that the 95 % of the injected MCEs Xen handler protects the target domain and it causes the panic of the target system. Only in the 5% of the errors is tolerated and it does not affect the correct behavior of the system. Moreover, when the same faultload is injected in guest context, i.e. the hypervisor sends to the guest the 5% of errors.

5.4.3 Campaign 2

The first campaign showed that also Xen handler is not able to differentiate among error codes, so a further campaign has been performed to explore its recovery behavior. In this campaign, rather than exhaustively trying different error codes, we used different combinations of the diagnostic information provided by the MCA The campaign encompassed 1536 MCEs injected when the bit SER_P is set, i.e., recovery actions are supported. Results reported in Figure 5.13 confirms that the target handler mainly provides coarse-grained recovery actions, i.e., system panic (70%) and process kill (10%). Again, the handler of the guest operating system does not implement any recovery action for the 20% of the errors.



(a) Hypervisor context

(b) Guest Context

Figure 5.12: Recovery Actions for Campaign #2

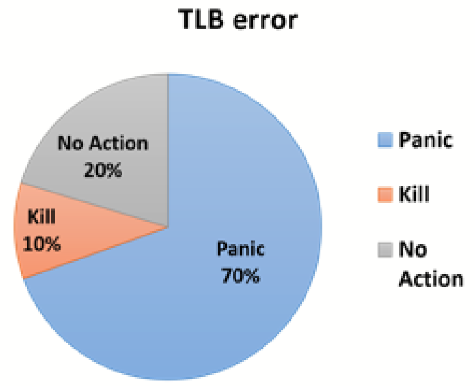


Figure 5.13: SER_P=1: Recovery Actions Implemented by Xen

5.5 Summary

This chapter presented fault injection framework developed for supporting dependability analysis of multicore systems. The approach leverages the notion of machine check error and the error-report mechanism implemented by modern processors. So far, fault injection campaigns have been conducted to test the functionalities of the framework under the Linux OS running on the top of the Intel i7 processor. Moreover, the fault injection framework was extended to virtualized systems in order to evaluate hardware error handling mechanisms implemented by the hypervisor that is the component responsible of providing to virtual machines an abstraction of hardware resources. The tool was developed for Xen hypervisor and exploits its MCA for error reporting architecture. Preliminary fault injection campaigns were conducted to test the ability of the tool to correctly inject MCEs. However, results revealed that Xen hypervisor implements only one recovery action, which is system panic.

Chapter 6

Conclusions and Future Work

Fault injection is a widely adopted technique for supporting dependability evaluation of OTS-based systems in safety-critical domains. This dissertation focuses on three fault injection techniques employed in systems that integrates OTS components, i.e. Software Fault Injection based on binary code mutation, Interface Error Injection (IEI) and SoftWare-Implemented Fault Injection (SWIFI). In particular, the focus was on the *quality* of existing techniques represented by *accuracy* and *representativeness* properties. Three different problems were addressed: (i) the accuracy of SFI based on code mutations; (ii) the representativeness of existing error models on which IEI techniques are based on; (iii) the representativeness and the effectiveness of existing SWIFI techniques when applied to systems deployed on multicore processors.

Concerning the accuracy of Software Fault Injection based on code mutation, this thesis investigates how code mutations are accurate to emulate software fault at binary level. Indeed, the injection at binary level implies the recognition of programming constructs by looking only at the binary code, which is a difficult and error-prone task. When injecting software faults at-binary level, a major concern is to assure that binary-level changes are accurately performed to emulate programming errors, i.e. SFI has to correctly emulate software faults to an acceptable degree of accuracy. Inaccuracies in the injections could negatively affect the results of fault injection campaigns leading to erroneously considerations on dependability properties of the system.

Firstly, this thesis describes an experimental evaluation of the accuracy of a binary fault injection technique based on code mutation, i.e. G-SWFIT. The evaluation is performed by comparing binary-level faults with source level faults generated by applying fault injection to complex real-world software system. Results revealed that several inaccuracies are related to the implementation of fault type constraints and to the identification of code blocks and control structures. These issues are not due to the G-SWFIT technique, and they can be avoided if an experimental evaluation of the fault injection tool is performed to improve the implementation. Based on the experimental results, this thesis proposed a systematic approach for testing and improving SFI based on binary code mutation. The approach automates the evaluation of SFI tools at injecting faults in binary code and it is based on the automatic generation of synthetic programs performed by an ad-hoc program generator named FaultProg. These synthetic programs represent a test suite for the BCM tool and the are generated with the sole purpose to evaluate the ability of the BCM tool to inject faults into them. The key idea is to control the generation of synthetic program, in such a way to expose the BCM tool to several different code patterns that could point out its limitations. In other words, the set of synthetic programs acts as a test suite for evaluating and improving binary-level fault injection and, more in general, mutation testing tools. Test-suite that can help to improve existing SFI tools at binary level, to develop SFI tools at binary level from the scratch which is the case when a new hardware architecture or different compilers are adopted.

Regarding Interface Error Injection (IEI), this dissertation investigates the representativeness of existing error models. IEI is a technique able to mimics the effects (i.e., errors) produced by faults in a component, by injecting exceptional or invalid values at the component's interface. Despite its popularity, the use of IEI for the representative emulation of component faults is questionable: there are not evidences that injecting errors at component interface is representative, i.e. mimic real software faults occurring in a system component and propagating to interfaces. Investigations on the representativeness of interface errors is required in order to perform an effective error injection based on representative error models.

This thesis provides an approach for analyzing how software faults in software components manifest as interface errors in order to provide some constructive evidence towards more representative IEI techniques. The experimental analysis aims at investigating errors propagation among components. To the aim, faults are injected in the software component under analysis by using a fault injection technique. Then, the faulty software component is instrumented and executed in order to identify the effects of injected faults on the system (i.e. program) that uses the component, including the corruption of data structures shared between the program and the component and erroneous return values from function calls. Results revealed that existing error models are not representative: (i) the corruption of single bit or byte is not enough to emulate the effect of a real software fault in the component since the data corrupted by injected faults include more than that; (ii) some corruptions should be performed by corrupting also memory area; (iii) the corruption rate of a memory address is strongly correlated to the number of access of that address during the fault-free execution, so it could be useful to corrupt the memory address more accessed during fault-free execution. Nevertheless, the proposed method was applied to few libraries considered as software components and future work will extend the method to analyze more complex software components.

Fault injection is also adopted to emulate hardware faults. This dissertation discusses the representativeness and the effectiveness of existing software-implemented fault injection (SWIFI) when used for dependability assessment of systems running on multicore. Based on bit stuck-at and bit-flip models, SWIFI allows the emulation of hardware faults through software by reproducing possible effects of real hardware errors without directly interfering with the system. Although these techniques were successfully employed for the evaluation of system behavior against hardware errors, two aspects should be considered when adopting existing fault models to multicore-based systems. First, single or multiple bit-flip could be automatically corrected by hardware mechanisms and masked to the software compromising the effectiveness of the SWIFI techniques. Second, new errors (that were not a concern in single-core architectures) may occur, e.g. errors in the interconnection links between cores, making existing SWIFI techniques not representative enough for modern processors.

This thesis proposes a fault injection framework developed for supporting dependability analysis of multicore-based systems. The approach leverages the notion of machine check error and the advanced error detection and reporting mechanisms implemented by modern processors. So far, fault injection campaigns have been conducted to test the functionalities of the framework under the Linux OS running on the top of the Intel i7 processor. Results showed that existing operating systems does not implement adequate mechanisms to cope with hardware errors, i.e. machine check errors, that are signaled by the error reporting architecture: recovery actions implemented by Linux OS are only system panic and process kill. Moreover, the approach was extended to virtualized systems deployed on multicore architecture since the adoption of this technology seems to be a good choice for the implementation of some solutions such as hardware and software consolidation. The approach aims at evaluating the error handling mechanisms of hypervisors (i.e. the software layer that implements the virtualization) in order to evaluate how hypervisors protect virtual machines when hardware errors occur. Preliminary esperiments were conducted to inject machine check errors in Xen hypervisor. Results showed that the only recovery action implemented in the hypervisor consists in system panic.

In the future, we will extend the framework by addressing the emulation of simultaneous errors affecting different cores, burst of errors, errors propagation among cores in both virtualized and not virtualized systems. The framework will be used to validate error handling of different operating systems, to analyze fault-tolerant mechanisms implemented across cores, to assess the resiliency of a given system under errors, and to benchmark the dependability behavior of different solutions adopting multicore.

Bibliography

[1] AUTOSAR Home Page. `http://www.autosar.org`.

[2] Critical Step. `http://www.critica-step.eu`.

[3] Intel 64 and IA-32 Architectures Software Developer's Manual Vol.3: System Programming Guide. `http://www.intel.com/`.

[4] MOBILAB Group. `http://www.mobilab.unina.it`.

[5] VMWare ESX and ESXi product. `http://www.vmware.com/products/esx/index.html`.

[6] WindRiver Hypervisor. `http://www.windriver.com/products/hypervisor/`.

[7] Hicham Agrou, Pascal Sainrat, M Gatti, David Faura, and Patrice Toillon. A design approach for predictable and efficient multi-core processor for avionics. In *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, pages 7D3–1. IEEE, 2011.

[8] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI: Generic Object-Oriented Fault Injection Tool. In *DSN*, pages 83–88, 2001.

[9] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411. ACM, 2005.

[10] 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*, 16(2):166–182, 1990.

[11] J. Arlat, J.C. Fabre, M. Rodríguez, and F. Salles. Dependability of COTS Microkernel-Based Systems. *IEEE*, 51(2):138–163, 2002.

[12] C Aussagues, D Chabrol, V David, D Roux, N Willey, A Tournadre, and M Graniou. Pharos, a multicore os ready for safety-related automotive systems: results and future prospects. *Proc. of The Embedded Real-Time Software and Systems (ERTS2)*, 2010.

[13] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE*, 1(1):11–33, 2004.

[14] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004.

[15] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. Fault Injection Experiments using FIAT. *IEEE*, 39(4):575–582, 1990.

[16] Victor R Basili and Barry Boehm. Cots-based systems top 10 list. *Computer*, 34(5):91–95, 2001.

[17] Alfredo Benso, Paolo Prinetto, Maurizio Rebaudengo, and M Sonza Reorda. Exfi: a low-cost fault injection system for embedded microprocessor-based boards. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 3(4):626–634, 1998.

[18] PG Bishop, RE Bloomfield, and PKD Froome. *Justifying the use of software of uncertain pedigree (SOUP) in safety-related applications.* University of Southampton, Institute of Sound and Vibration Research, 2001.

[19] Pearl Brereton and David Budgen. Component-based systems: A classification of issues. *Computer*, 33(11):54–62, 2000.

[20] Bryan Turner. The Random C Program Generator. `https://sites.google.com/site/brturn2/randomcprogramgenerator`, 2008.

[21] J. Carreira, H. Madeira, and J.G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE*, 24(2):125–136, 1998.

[22] Joao Carreira, Henrique Madeira, and Joao Gabriel Silva. Xception: Software Fault Injection and Monitoring in Processor Functional Units. In *DCCA*, pages 135–149, 1995.

[23] Subhachandra Chandra and Peter M Chen. How fail-stop are faulty programs? In *FTCS*, pages 240–249. IEEE, 1998.

[24] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.Y. Wong. Orthogonal Defect Classification–A Concept for In-Process Measurements. *IEEE*, 18(11):943–956, 1992.

[25] J. Christmansson and R. Chillarege. Generation of an Error Set that Emulates Software Faults based on Field Data. In *FTCS*, pages 304–313, 1996.

[26] J. Christmansson, M. Hiller, and M. Rimen. An Experimental Comparison of Fault and Error Injection. In *ISSRE*, pages 369–378, 1998.

[27] International Electrotechnical Commission et al. Functional safety of electrical/electronic/programmable electronic safety-related systems-part 2: Requirements for electrical/electronic/programmable electronic safety systems. *CEI/IEC*, pages 61508–2, 2000.

[28] D. Cotroneo and R. Natella. Fault Injection for Software Certification. *IEEE*, 11(4):38–45, 2013.

[29] Domenico Cotroneo, Anna Lanzaro, Roberto Natella, and Ricardo Barbosa. Experimental Analysis of Binary-Level Software Fault Injection in Complex Software. In *EDCC*, pages 162–172, 2012.

[30] M. Daran and P. Thévenod-Fosse. Software Error Analysis: A Real Case Study Involving Real Faults and Mutations. *ACM Soft. Eng. Notes*, 21(3):158–171, 1996.

[31] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE*, pages 733–752, 2006.

[32] US DOE. System safety program requirements, us department of defense. Technical report, MIL-STD-882D, 2000.

[33] Heradon Douglas and Christian Gehrmann. Secure virtualization and multicore platforms state-of-the-art report. *Swedish Institute of Computer Science*, 2009.

[34] J. Durães and H. Madeira. Emulation of Software Faults by Educated Mutations at Machine-Code Level. In *ISSRE*, pages 329–340, 2002.

[35] J. Durães and H. Madeira. Generic Faultloads based on Software Faults for Dependability Benchmarking. In *DSN*, pages 285–294, 2004.

[36] J.A. Durães and H.S. Madeira. Emulation of Software faults: A Field Data Study and a Practical Approach. *IEEE*, 32(11):849–867, 2006.

[37] Sigrid Eldh, Sasikumar Punnekkat, Hans Hansson, and Peter Jönsson. Component Testing Is Not Enough–A Study of Software Faults in Telecom Middleware. In *Testing of Softw. and Comm. Sys.*, pages 74–89. 2007.

[38] C. Esposito, D. Cotroneo, and N. Silva. Preliminary investigation on safety-related standards. Technical report, 2011. Technical Report www.mobilab.unina.it/techreports.html.

[39] European Cooperation for Space Standardization. ECSS-E-70-41A – Ground Systems and Operations: Telemetry and Telecommand Packet Utilization, 2003.

[40] Iman Faraji, Moslem Didehban, and Hamid R Zarandi. Analysis of transient faults on a mips-based dual-core processor. In *Availability, Reliability, and Security, 2010. ARES'10 International Conference on*, pages 125–130. IEEE, 2010.

[41] Karl-Filip Faxén, Christer Bengtsson, Mats Brorsson, Håkan Grahn, Erik Hagersten, Bengt Jonsson, Christoph Kessler, Björn Lisper, Per Stenström, and Bertil Svensson. Multicore computing–the state of the art. 2008.

[42] Thomas K Ferrell and Uma D Ferrell. Rtca do-178b/eurocae ed-12b. *The Avionics Handbook*, 2000.

[43] J. Flich, S. Rodrigo, J. Duato, T. Sodring, A.G. Solheim, T. Skeie, and O. Lysne. On the potential of noc virtualization for multicore chips. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, 2008.

[44] Thomas Gaska, Brian Werner, and David Flagg. Applying virtualization to avionics systems the integration challenges. In *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*, pages 5–E. IEEE, 2010.

[45] Thomas Gaska, Brian Werner, and David Flagg. Applying virtualization to avionics systems—The integration challenges. In *Digital Avionics Systems ...*, pages 1–19, 2010.

[46] GCC Online Documentation. Options for Debugging Your Program or GCC. `http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html`, 2014.

[47] A.K. Ghosh, M. Schmid, and V. Shah. Testing the Robustness of Windows NT Software. In *ISSRE*, pages 231–235, 1998.

[48] Anup K Ghosh and Matthew Schmid. An approach to testing COTS software for robustness to operating system exceptions and errors. In *ISSRE*, pages 166–174. IEEE, 1999.

[49] J. Gray. A Census of Tandem System Availability between 1985 and 1990. *IEEE*, 39(4):409–418, 1990.

[50] Ulf Gunneflo, Johan Karlsson, and Jan Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*, pages 340–347. IEEE, 1989.

[51] S. Han, KG Shin, and HA Rosenberg. DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment. In *IPDS*, pages 204–213, 1995.

[52] Gernot Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 11–16. ACM, 2008.

[53] M. Hiller, A. Jhumka, and N. Suri. EPIC: Profiling the propagation and effect of data errors in software. *IEEE*, 53(5):512–530, 2004.

[54] JJ Hudak, B.H. Suh, DP Siewiorek, and Z. Segall. Evaluation and Comparison of Fault-Tolerant Software Techniques. *IEEE*, 42(2):190–204, 1993.

[55] James W Hunt and Thomas G Szymanski. A fast algorithm for computing longest common subsequences. *Comm. ACM*, 20(5):350–353, 1977.

[56] International Organization for Standardization. Product development: software level. *ISO 26262-6*, 2012.

[57] Gabriela Jacques-Silva, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Dependability assessment of operating systems in multi-core architectures. In *Fast Abstract in the 38th Int. Symp. on Dependable Systems and Networks, Anchorage, Alaska (June 2008)*, 2008.

[58] T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun, and T. Marteau. Analysis of the Effects of Real and Injected Software Faults: Linux as a Case Study. In *PRDC*, pages 51–58, 2002.

[59] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE*, 37(5):649–678, 2011.

[60] A. Jin and J. Jiang. Fault Injection Scheme for Embedded Systems at Machine Code Level and Verification. In *PRDC*, pages 55–62. IEEE, 2009.

[61] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE*, 44(2):248–260, 1995.

[62] K. Kanoun and L. Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society, 2008.

[63] W.-I. Kao and R.K. Iyer. DEFINE: A Distributed Fault Injection and Monitoring Environment. In *FTPDS*, pages 252–259, 1994.

[64] W.-I. Kao, R.K. Iyer, and D. Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults. *IEEE*, 19(11):1105–1118, 1993.

[65] Larry M Kinnan. Use of multicore processors in avionics systems and its potential impact on implementation and certification. In *Digital Avionics Systems Conference, 2009. DASC'09. IEEE/AIAA 28th*, pages 1–E. IEEE, 2009.

[66] Andi Kleen. Machine check handling on linux. *SUSE Labs*, 2004.

[67] P. Koopman and J. DeVale. The Exception Handling Effectiveness of POSIX Operating Systems. *IEEE*, 26(9):837–848, 2000.

[68] Kirk L Kroeker. The evolution of virtualization. *Communications of the ACM*, 52(3):18–20, 2009.

[69] Rikard Land, Laurens Blankers, Michel Chaudron, and Ivica Crnković. Cots selection best practices in literature and in industry. In *High Confidence Software Reuse in Large Systems*, pages 100–111. Springer, 2008.

[70] Dongwoo Lee and Jongwhoa Na. A novel simulation fault injection method for dependability analysis. *IEEE Design & Test of Computers*, 26(6):0050–61, 2009.

[71] Peter Lindsay and Graeme Smith. Safety assurance of commercial-off-the-shelf software. Technical report, 2000.

[72] H. Madeira, D. Costa, and M. Vieira. On the Emulation of Software Faults by Software Fault Injection. In *DSN*, pages 417–426, 2000.

[73] P.D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Computer Sys.*, 29(4):11:1–11:38, 2011.

[74] E. Martins, C.M.F. Rubira, and N.G.M. Leme. Jaca: A Reflective Fault Injection Tool based on Patterns. In *DSN*, pages 483–487, 2002.

[75] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[76] B Craig Meyers and Patricia Oberndorf. *Managing software acquisition: Open systems and COTS products*. Addison-Wesley Longman Ltd., 2001.

[77] R. Moraes, R. Barbosa, J. Durães, N. Mendes, E. Martins, and H. Madeira. Injection of Faults at Component Interfaces and Inside the Component Code: Are They Equivalent? In *EDCC*, pages 53–64, 2006.

[78] R. Moraes, J. Durães, R. Barbosa, E. Martins, and H. Madeira. Experimental Risk Assessment and Comparison using Software Fault Injection. In *DSN*, pages 512–521, 2007.

[79] Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. In *Design and Test Workshop (IDT), 2011 IEEE 6th International*, pages 12–17. IEEE, 2011.

[80] Roberto Natella, Domenico Cotroneo, Joao A Durães, and Henrique S Madeira. On Fault Representativeness of Software Fault Injection. *IEEE*, 39(1):80–96, 2013.

[81] Nicolas Navet, Bertrand Delord, Markus Baumeister, et al. Virtualization in automotive embedded systems: an outlook. In *Seminar at RTS Embedded Systems*, 2010.

[82] Nicolas Navet, Aurélien Monot, Bernard Bavoux, and Françoise Simonot-Lion. Multi-source and multicore automotive ecus-os protection mechanisms and scheduling. In *Industrial Electronics (ISIE), 2010 IEEE International Symposium on*, pages 3734–3741. IEEE, 2010.

[83] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM Sigplan Not.*, 42(6):89–100, 2007.

[84] Wee Teck Ng and Peter M Chen. The design and verification of the rio file cache. *IEEE*, 50(4):322–337, 2001.

[85] W.T. Ng and PM Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *FTCS*, pages 76–83, 1999.

[86] Patricia Oberndorf. Cots and open systems. 1998.

[87] D. Oppenheimer, A. Ganapathi, and D.A. Patterson. Why Do Internet Services Fail, and What Can Be Done About It? In *USITS*, 2003.

[88] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *CGO*, pages 2–11. ACM, 2010.

[89] William J Perry. Specifications and standards-a new way of doing business. *US Department of Defense Policy Memorandum*, 1994.

[90] P. Popov and L. Strigini. Assessing Asymmetric Fault-Tolerant Software. In *ISSRE*, pages 41–50, 2010.

[91] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield, Dan Magenheimer, Jun Nakajima, and Asit Mallick. Xen 3.0 and the art of virtualization. In *Linux Symposium*, page 65, 2005.

[92] K. Qian, D. Haring, and L. Cao. *Embedded Software Development with C*. Springer, 2009.

[93] Radisys.

[94] Manuel Rodríguez, Frédéric Salles, Jean-Charles Fabre, and Jean Arlat. Mafalda: Microkernel assessment by fault injection and design aid. In *Dependable Computing - EDCC-3*, pages 143–160. Springer, 1999.

[95] Felix Salfner, P Troger, and Steffen Tschirpke. Cross-core event monitoring for processor failure prediction. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*, pages 67–73. IEEE, 2009.

[96] B.P. Sanches, T. Basso, and R. Moraes. J-SWFIT: A Java Software Fault Injection Tool. In *LADC*, 2011.

[97] Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V Adve. mswat: low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 122–132. ACM, 2009.

[98] David J Sheskin. *Handbook of parametric and nonparametric statistical procedures*. Chapman and Hall/CRC, 2003.

[99] Daniel Skarin, Raul Barbosa, and Johan Karlsson. Goofi-2: A tool for experimental dependability assessment. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 557–562. IEEE, 2010.

[100] ARINC Specification. 651: Design guidance for integrated modular avionics, ser. *ARINC report. Airlines Electronic Engineering Committee (AEEC) and Aeronautical Radio Inc*, 1991.

[101] Def Stan. Stan 00-55. *The Procurement of Safety Related Software in Defence Equipment-Parts*, 1:00–55, 1991.

[102] Def Stan. Stan 00-56. *Safety Management Requirements for Defence Systems, UK Ministry of Defence, Defence Standard 00-56*, (2), 1996.

[103] D.T. Stott, B. Floering, Z. Kalbarczyk, and R.K. Iyer. A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *IPDS*, page 91, 2000.

[104] Sang-Bum Suh, Joo-Young Hwang, Joon-Young Shim, JaeMin Ryu, Sungkwan Heo, ChanJu Park, ChulRyun Kim, Jae-Ra Lee, Ilpyoung Park, and Hosoo Lee. Computing state migration between mobile platforms for seamless computing environments. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 1216–1217. IEEE, 2008.

[105] M. Sullivan and R. Chillarege. Software Defects and their Impact on System Availability: A Study of Field Failures in Operating Systems. In *FTCS*, pages 2–9, 1991.

[106] J Swingler and JW McBride. The synergistic relationship of stresses in the automotive connector. 1998.

[107] Clemens Szyperski. *Component software: beyond object-oriented programming.* Pearson Education, 2002.

[108] Sriraman Tallam, Chen Tian, Rajiv Gupta, and Xiangyu Zhang. Enabling Tracing of Long-Running Multithreaded Programs Via Dynamic Execution Reduction. In *ISSTA*, pages 207–218. ACM, 2007.

[109] Dong. Tang and H. Hecht. An Approach to Measuring and Assessing Dependability for Critical Software Systems. In *ISSRE*, pages 192–202. IEEE, 1997.

[110] S.K. Thompson. Sample Size for Estimating Multinomial Proportions. *The American Statistician*, 41(1):42–46, 1987.

[111] T.K. Tsai and R.K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *MMB*, 1995.

[112] Marco Vieira and Henrique Madeira. A dependability benchmark for OLTP application environments. In *VLDB*, pages 742–753. VLDB Endowment, 2003.

[113] Mark Vigder and John Dean. An architectural approach to building systems from cots software components. 1997.

[114] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly “Good” Software Can Behave. *IEEE*, 14(4):73–83, 1997.

[115] J.M. Voas. Certifying off-the-shelf software components. *IEEE*, 31(6):53–59, 1998.

[116] Elaine J Weyuker. Testing component-based software: A cautionary tale. *IEEE*, 15(5):54–59, 1998.

[117] Wind River. Applying Multi-core and Virtualization to Industrial and Safety-Related Applications.

[118] S. Winter, C. Sârbu, N. Suri, and B. Murphy. The impact of fault models on software robustness evaluations. In *ICSE*, pages 51–60. ACM, 2011.

[119] Yang-Ming Zhu and Steven M Cochoff. Medical image viewing on multicore platforms using parallel computing patterns. *IT professional*, 12(2):33–41, 2010.