Università degli Studi di Napoli Federico II
Dottorato in Ingegneria Informatica ed Automatica
Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione (DIETI)
ING-INF/05

Robustness Evaluation of Software Systems through Fault Injection

Candidate
Domenico Di Leo

Supervisor
Prof. Domenico Cotroneo

PhD Coordinator
Prof. Franco Garofalo

ciclo XXV, 2010-2013

Università degli Studi di Napoli Federico II, Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione (DIETI).

Never Give Up!

Acknowledgments

First of all, I want to express my gratitude to my advisor prof. Domenico Cotroneo for giving me this opportunity and to prof. Johan Karlsson for hositng and mentoring me while I was a visitor scholar at Chalmers University (Sweden).

I am thankful to Paolo Salvatore and Marco Romeo from Ciaotech s.r.l. for financing my graduate studies and introducing me to another relevant aspect of the research: its management and financing.

I especially want to thank the "Robs", Roberto Natella and Roberto Pietrantuono, for their continuous support, fruitful discussions and fundamental collaboration in several works. Sincerely, I consider them as my second advisor and wish them a brilliant and outstanding career.

Thanks to Francesco Fucci for his valuable support and intense discussion during the preparation of a joint work.

I owe many thanks to all other Mobilab group members for their incitement.

The last but not the least, I sincerely thank my dear family for being on my side and encouragement especially in this last period of my life... Thank you!

Domenico Di Leo

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last decades, software has been introduced in desperate safety domains, such as automotive, avionics and railways, just to name a few. For these domains, software is demanded to be highly dependable since its failure may endanger human life, harm the environment, or cause economical loss. A growing number of safety-critical operational functions traditionally hardware implemented have been accounted to software. The avionics domains offers a clear example with the shift from a federated architecture, in which each Electronic Component Unit (ECU) delivers a specific task, to a central architecture where a single software based system executes tasks of mixed criticality. As a consequence, the complexity and the volume of software is steadily growing in safe-mission critical systems.

In 2004, the American Department of Defence (DoD) reports that "functionality provided by software for aircraft, has increased from about 10 percent in the early 1960s for the F-4 to 80 percent for theF/A-22" [1] while the lines of code (generally regarded as a measure for the size of software) has exceed over 5 million in modern jet, compared to about 1 million lines of code in older aircrafts [2]. Additionally, due to the pressure from the market, there is an emergent trend to integrate Commercial Off The Shelf software (COTS) or more in general OTS software in safety critical systems. Indeed, COTS software, that is, a software that is not developed in the project, rather, it is acquired from a vendor and used as-is or with minor modifications [3], it can potentially be a viable alternative to in-house software also for safety-critical systems [4].

Furthermore, demanding and severe standards that regulate the devel-

opment of software in safety domains allow to include COTS in the final released software product [5–8]. If from one side COTS products have the potentiality to reduce significantly the time to market, from another side a thorough and presumably costly safety assessment is preliminary to their integration. Whatever the nature (in-house or COTS), the complexity and the size, software must ensure that its behavior does not harm the system in which is part of as well as it must be *robust* to exceptional inputs coming from the surrounding environment. In other words, as safety standards recommend [5,7,8] **software components are elements of a larger systems, hence they have to operate correctly even in presence of software and hardware faults or exceptional conditions**. Software faults, also refereed as bugs, are not avoidable in practice given the complexity of software components [9]. Hardware faults will continue to increase due to technology scales and transistor wear out [10, 11].

Robustness failures due to software faults and hardware faults have been the cause of clamorous accidents and costly service disruption. For instance, in the infamous Ariane 5 disaster, the software reused from Ariane 4 proved to have robustness problems when operating in the conditions experienced from Ariane 5 [12]. Sun Microsystems found that cosmic ray strikes, a common cause of transient hardware errors, insisting on L2 cache with defective protection provoked the Sun servers to suddenly and mysteriously crash. Many large companies have been affected from this failure, among them Ebay [13]. As a consequence, a software component should be robust against erroneous behavior of faulty software components with which interact and, at the same time, should be tolerant to the faults due the hardware on which execute.

Fault injection, the deliberate inoculation of faults, is a powerful means to assess the robustness of software components that goes far beyond traditional testing techniques. In the last decades several fault injection technique have been proposed and evaluated. First fault injection approaches, the so called hardware implemented fault injection, introduced physical faults through the hardware layer of the target system (e.g., pin level injection [14]) or with an external injector (e.g., heavy ion radiation and electromagnetic interference source [15]). Because of the rapid increase in processor complexity, hardware implemented fault injection has been replaced with the Software Implemented Fault Injection (SWIFI). SWIFI emulates hardware faults through the software layer (e.g., insertion of fault injection code in

exception handling routine [16]) or with debugging functionalities available on modern processors [17].

SWIFI emulates stuck-at bit errors or transient errors that are representative of errors due to real hardware faults, however this techniques is not suitable for the injection of software faults. In this case, the Robustness Testing (RT) which injects errors (due to software faults) to the software interface of the target component is a more adequate technique. RT specifically selects exceptional values (via typical software testing methods) to inject into the application programming interface of the target. SWIFI and RT are widely used and safety standards recommend their application. For instance, the 20 years old DO178B [5] and it is newer version, DO178C [6] released in 2011, both used in the avionics, demand the RT. In 2011 the International Standard Organization (ISO) published the safety standard ISO26262 [8] for the automotive domain which refers to the RT with the name "interface test", whilst SWIFI is named "fault injection test".

These two techniques do not need to access to the source code of the target, hence they are suited for the assessment of COTS software that might be available only as an executable. Because COTS are not usually designed and developed according to safety standards, their behavior may be unreliable in presence of hardware and software faults. As a consequence, an assessment conducted with SWIFI and RT is highly recommended. Both RT and SWIFI have assessed the robustness of a variety of software components, such as distributed systems [18–20] and operating systems [21, 22, 22]. Despite the intensive use of these techniques, their application is still costly. Experienced personnel and several days or months are necessary to carry on "exhaustive" fault injection campaigns [23, 24].

More in general, the cost of software verification often exceeds half the overall cost of software development and maintenance [25]. Besides being costly, fault injection may contribute to dramatic economic loss if it is poorly or negligently performed. NIST reports that in the United States alone, the U.S. lost around 60B$ as a result of inadequate software testing in 2001. Therefore, **approaches and methods are sought to keep fault injection effective without compromising its efficiency**. Many factors affect the efficiency and the effectiveness such as the complexity of the injector, the number of faults to inject and the presence of a workload, i.e., an application which interacts with the target of the fault injection. The workload plays a significant dual role in fault injection. On the one hand it can facilitate

the activation of faults or drive the target in specific states, thus allowing fault injection to be effective. On the other hand, it must be chosen with care: the desire to use more workloads in order to augment the probability of activating a fault would make the fault injection inefficient.

This thesis focuses on robustness testing and software implemented fault injection, for both analyzes the effect of workload on the experiment outcomes. Furthermore, the thesis suggests approaches to make the fault injection techniques more cost-effective by leveraging on the workload.

1.1 Thesis Contribution

A wide literature exists on robustness testing [19, 21, 22, 26, 27] and software implemented fault injection [15, 17, 18, 28, 29], many of these studies have concentrated on the representativeness of the faults/errors to inject (*what to inject?*) and the location (*where to inject?*) or have assessed software fault tolerance mechanisms. For the robustness testing the error is selected through common software engineering methods (e.g., boundary value analysis) while the application program interface of the software component is the target location (e.g., in a UNIX system the POSIX interface). For the software implemented fault injection there is the acceptance of the bit flip (the temporary permutation of a bit) as representative of hardware faults occurring in the memory area and the board registers (e.g., general purpose registers). However, there is a marginal investigation [30–32] on how through the workload the fault injection can be more effective and at the same time to keep fault injection efficient in terms of number of injections to execute. Hence, in this thesis we investigate the effect of the workload on the RT and SWIFI. More specifically we address the following questions:

- **Does and how the workload influence the outcomes of RT?** The workload clearly effects the execution of the target especially when stimulates complex systems (e.g., an operating system or a middleware) which have different *states*. For instance, a workload running on an operating systems that performs I/O operations brings the OS in a state that differs from the one due to a CPU intensive workload. This aspects is relevant if we consider that robustness vulnerability are characterized by rare and subtle activation conditions. Therefore, the workload can potentially improve the efficiency of RT because it

can activate such conditions. In this thesis, we focus on the operating system (OS) because they are one of the major COTS component used in safety critical systems in avionics [33]. In particular, the case study is an industrial OS, FIN.X-RTOS, developed in the context of a pilot R&D project, in conjunction with Finmeccanica s.p.a. From results of this thesis, an important emerging aspect is that, to improve the effectiveness of robustness testing, fault injection campaigns should consider one more variables, other than exceptional inputs; that is, the current state of the OS.

By Combining the workload with the RT is possible to conceive a new testing strategy, that is, a **stateful robustness testing** that outperforms traditional RT (stateless RT) and stress testing. Indeed, we observed a larger number of failures at application level and an increment of the statement coverage up to 15% compared to RT. More importantly the increment occurs in parts of code that are hard to cover. Although we recognize that more appropriated test strategies exist for augmenting the coverage (e.g, evolutionary testing), stateful RT can be a complementary and relative simple technique to adopt when portions of code are tough to cover.

- **How can we include the workload in RT?** Obviously, the workload cannot be left unspecified if it influences the outcome of the RT. As said, robustness testing should be extended in the stateful robustness testing. The vexing challenge is to propose approaches that can model the state of the OS. Traditional approaches derived from objected oriented software do not scale well for the OS since it is the result of a complex and intricate design. Additionally such approaches are not feasible if there is no specification as it is often the case for open source OSs. Thus, we conceived two alternative approaches for modeling the file system, the component of the OS under test. In the first approach, the model of the file systems, manually created, encompasses entities a file system is composed of, and resources it uses contributing to determine its state. The second approach, named StAte-Based Robustness testIng of operatiNg systEms, SABRINE, *automatically* derives behavioral models from execution traces of the file system and executes the test cases. Both approaches have been experimented on FIN.X-RTOS and can be adopted for any operating system.

- **Does and how the workload influence the outcomes of SWIFI?** Undoubtedly, the workload is responsible for the activation of faults and therefore impacts the outcome of a fault injection campaign. If we think of an application running on the bare metal to which to provide a set of inputs, we are likely to observe variation in the failure distribution because inputs activate different execution paths in the workload. This dissertation illustrates for a set of OTS applications running on a PowerPc board, yet representative of real workloads, the relationship between the characteristics of the input (e.g., its size) and the failure distribution. Results show that the size of the input data can induce a fluctuation of about 30% of the percentage of value failure (silent data corruption).

- **How can we make RT and SWIFI more cost-effective?** Intuitively the presence of the workload can potentially increase the number of fault injection campaign. Indeed, if we accept that a workload drives the target in different states, we should conduct RT for each one. Similarly when injecting hardware faults with SWIFI, we should evaluate the robustness of the workload for each point of the input domain. These considerations would make the cost of the techniques unsustainable. Therefore, in this dissertation we present possible solutions to keep the application of these techniques cost-effective without compromising their efficiency.

  SABRINE, through clustering techniques, can keep the number of test cases limited and when compared with Random testing (a common baseline) can achieve the same results with a test suite two order of magnitude smaller. This thesis also shows that SWIFI does not need to target the workload for each input provided, rather it is possible to reduce the number of fault injection campaigns by clustering the input domain before injecting faults. The approaches is straightforward and for specific applications in our case study allows to reduce the number of fault injection by 45%.

The dissertation is organized as follows: Chapter 2 illustrates basic concepts on robustness and its evaluation through fault injection techniques. Chapter 3 provides the background on robustness testing and surveys previous revelent works with a specific focus on the application to operating systems. Chapter 4 is fully devoted to the description of a novel approach which

extends robustness testing into statefull robustness testing. Chapter 5 provides the basic concepts on hardware fault injection and focuses especially on SWIFI. Chapter 6 discusses the results on an investigation between workload inputs and failure distribution. Chapter 7 concludes with final remarks, the indication of the lesson learned and future research directions.

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

- D. Cotroneo, **D. Di Leo**, R. Natella, R. Pietrantuono, *A Case Study on State-Based Robustness Testing of an Operating System for the Avionic Domain*, Proc. of the 30th International Conference on Computer Safety, Reliability and Security (SAFECOMP), September 2011, Naples, Italy

- **D. Di Leo**, B. Sanghoolie, F. Ayat, J. Karlsson, *On the impact of hardware faults on embedded computer systems- An investigation of the relationship between workload input and failure mode distributions*, Proc. of the 31th International Conference on Computer Safety, Reliability and Security (SAFECOMP), September 2012, Magdeburg, Germany

- D. Cotroneo, **D. Di Leo**, F. Fucci, R. Natella, *SABRINE: StAte-Based Robustness testIng of operating systems*, submitted to the ACM International Symposium in Software Testing and Analysis (ISSTA), 2013

- D. Cotroneo, **D. Di Leo**, R. Natella, *Adaptive Monitoring in Microkernel OSs*, DSN Workshop on Proactive Failure Avoidance, Recovery and Maintenance (PFARM), Proc. of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W) June 2010, Chicago, Illinois, USA

- D. Cotroneo, **D. Di Leo**, N. Silva, R. Barbosa, *The PreCertification Kit for Operating Systems in Safety Domains*, Software Certification (WoSoCER), 2011 First International Workshop on , vol., no., pp.19-24, Nov. 29 2011-Dec. 2 2011

Chapter 2

Robustness

This chapter provides the terminology and the basic concept on which lay the entire thesis. Firstly, we introduce the concept of robustness which is a property or attribute of a software system that have been formalized in both dependability engineering and software engineering. Then, the focus shift on one largely adopted technique for robustness evaluation: fault injection.

2.1 Basic Concepts and Definitions from Dependability

A system is an entity, including human being, software and hardware, that implements specific functionalities. A system interacts with its surrounding environment to which provide services through the system boundary. A system can consist of interacting components, recursively each component is itself made of components. The ultimate component is the atomic unit: any further internal structure cannot be discerned, or is not of interest and can be ignored. One or more services of the system can fail. More specifically (see Figure 2.1):

- **Failures** or service failures are 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

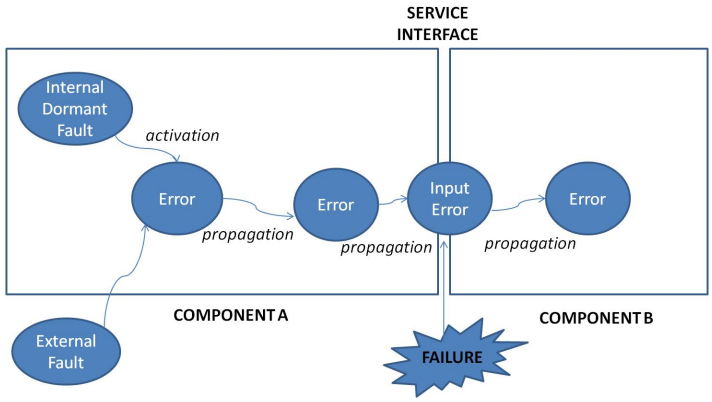

Figure 2.1: Fault-Error-Failure chain.

one or more services implementing system functions make the system operate in a degraded mode.

- **Errors** are 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.

- **Faults** are the hypnotized cause of errors. They can be of different types: development faults that occur during the system development, physical faults including faults related to hardware and, human interaction faults due to the interaction with the components or the system. An active faults becomes an error, while a *latent* fault is present in the component or in the system but it has not been manifested as an error. Hence, the only presence of faults is not sufficient for a failure, but it is necessary their activation that may occur under specific triggering conditions. For instance, a fault is activated only when the software component is in a specific state or the hardware executes some instructions. Therefore, given the activation conditions, faults can be further classified as: Bohrbugs which have simple and deterministic activation conditions, Heisenbugs are non deterministic are difficult (if no impossible) to reproduce, Mandelburgs which have deterministic activation but they require a complex conditions to activate. At the

present, there is no agreement between the definitions of Mandelburgs and Heisengburg that can be defined in a different manner considering other factors (such as the delay for their activation). Faults can also be classified as *external* and *internal*. The prior presence of a vulnerability, i.e., an internal fault that enables an external fault to harm the system, is necessary for an external fault to cause an error and possibly subsequent failure(s).

A system is said to be *dependable* if it can avoid service failures that are more frequent and more severe than is acceptable. **Dependability** embraces the following primary 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.

Robustness is defined as the specialization of the primary attributes, i.e., **the dependability with respect to external faults**.

2.2 Basic Concepts and Definitions from Software Engineering

Robustness expresses *the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions* [34]. Similarly to the dependability community, in the software engineering the definition of robustness is related to external stimuli. However, if in dependability engineering a robustness failure may occur because of an internal component error, in software engineering the word error is meant as the mistake that makes a human being (e.g., a programmer) to introduce a fault in the software and the fault is the cause of the failure. For the sake of completeness, we provided definitions and concepts current in use in software engineering, however in this works we consistently adopt the definitions in Section 2.1.

2.3 Robustness Evaluation

As the robustness of a software component has repercussions on the dependability of the whole system, it is essential to evaluate it. Robustness evaluation is indispensable for COTS software because they have been developed by ignoring a specific application domain. COTS software is more and more integrated in safety critical systems [33] hence they must react robustly when exposed to unexpected inputs coming from hardware as well as from software. There are several means to this end, from modeling (e.g., formal methods) to testing. Fault Injection (FI) is a common verification technique for the assessment of COTS software [4, 19, 22, 30, 35].

2.3.1 Basic Concepts and Definition on Fault Injection

Fault injection (FI) is extensively adopted for evaluating the robustness of software components or their ability to handle faults, in other words to be fault tolerant. With the support of FI is possible to:

- Test the effectiveness of fault-handling mechanisms.
- Study error propagation and error latency.
- Verify failure mode assumptions.
- Extract data to serve as inputs for modeling (e.g, reliability models).

FI can target either *real systems* or *models of the system*. By real systems, we mean an actual implementation of the system either commercial or prototypal. System model for fault injection can be of two different types *software simulation* and *hardware emulation*. Software simulation fault injection can be carried out on a simulator of the system at several different abstractions. Hardware emulation fault injection is based on model of the system implemented with large Field Programmable Gate Array (FPGA) circuits. These models can provide a detailed representation of the relative real system.

We introduce concepts and definitions that are valid regardless of the specific technique. The system to be assessed is named **target system** or simply **target**. The target executes a **workload**, an application or program that stimulates the target. An **experiment** consists in the injection of a single fault into the target. The result of an experiment is also named

outcome and represent how the target reacts to the fault. A **campaign** is a collection of experiments. The set of faults injected during a campaign represent the **faultload**. An execution of the workload in absence of faults (fault-free) is called **golden run** and its result is usually compared with the results of fault injection in order to detect failures. A **fault model** conventionally describes the faults in terms of their type, location and trigger. In other words, the fault model indicates the nature of the fault (what to inject?), the service or the location in which to inject the fault (where to inject?) and the time interval/instant in which to inject (when to inject?). Related to the triggering is also the duration of the injection (how long to inject?). These aspects will be detailed when discussing the RT (Chapter 3) and SWIFI (Chapter 5).

Fault injection techniques present the following characteristics [36]:

- **Controllability** - ability to control the injection of faults in time and space.
- **Observability** - ability to observe and record the effects of an injected fault.
- **Repeatability** - ability to repeat a fault injection experiment and obtain the same result.
- **Reproducibility** - ability to reproduce the results of a fault injection campaign.
- **Reachability** - ability to reach possible fault locations inside an integrated circuit, or within a program.
- **Fault Representativeness** - how accurately the faultload represents real faults.
- **Workload Representativeness** - how accurately the workload represents real system usage.
- **System Representativeness** - how accurately the target system represents the real system.

Either software simulation or hardware emulation allows to inject faults in a manner more accurate compared to injection into real systems, on the opposite, a higher representativeness is achieved when injecting in real systems. However, the cost to develop the simulator and the simulation may be

high to prevent software simulation or hardware emulation from their use. Controllability, observability, repeatability, and reproducibility are higher in software simulation and hardware emulation than in fault injection into real systems.

2.3.2 A Brief Overview on Fault Injection Techniques

In this section, we group fault injection techniques applied to real systems according to the fault model and their implementation[1], see Fig. 2.2.

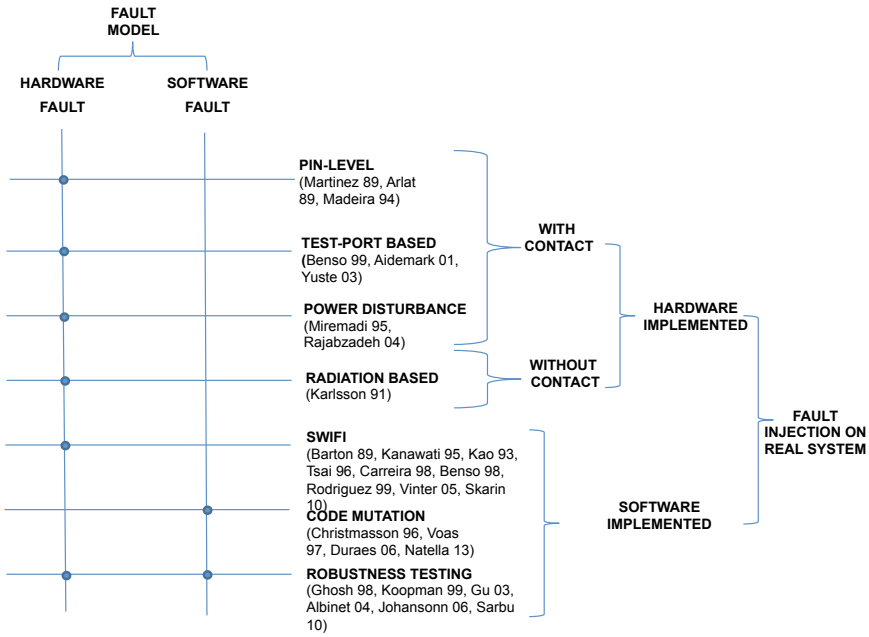

Figure 2.2: An overview of fault injection techniques classified by fault model and their implementation. Under the fault injection technique appear the names of the main contributors along with the year in which the technique has been proposed/refined.

Early fault injection experiments aimed to simulate hardware faults, that is, emulating malfunctioning hardware. Techniques that emulates hardware faults are classified as **hardware implemented** and **software imple-**

[1] additional details are provided in Chapter 3 and Chapter 5

mented. Hardware implemented fault injection requires a dedicated hardware for the injection that can occur with contact to the target or without the contact (radiation). Techniques that inject software faults can be dived into code mutation and in robustness testing. Code mutation performs fault injection on source code which is literally manipulated. Robustness testing injects faults into the interface of the target and can emulate either hardware faults or software faults.

Chapter 3

Robusteness Testing

Robustness testing is one of the major technique adopted for robustness assessment of software components against software faults. Along the years, it has been applied to several types of software systems from operating systems to web applications. This chapter narrows the application of robustness testing to operating systems which are a class of software widely spread in safety domains and one of them is the case study of this thesis.

3.1 Robustness Testing Approaches

As explained in Section 2.1 robustness is evaluated in presence of the errors that originate from faults.

This means, with reference to the Figure 3.1(a), that faults are injected into the component A and if they become active, there is a chance they manifest at the interface of the component A as errors. Then these errors may propagate to component B which is the target of the RT. The injection of faults in A ia achieved through code mutation. This technique has been adopted in [37], although it is effective, it requires that injected faults are representative of real faults [38], their activation and their propagation to component B interface. Another possible choice is to inject errors directly at the interface of the target, component B in Figure 3.1(b). In practice, the parameters of the service (a function) are corrupted with specific errors. This approach compared to the former one does not require the activation of a fault, but it is sufficient to observe a service invocation towards the target.

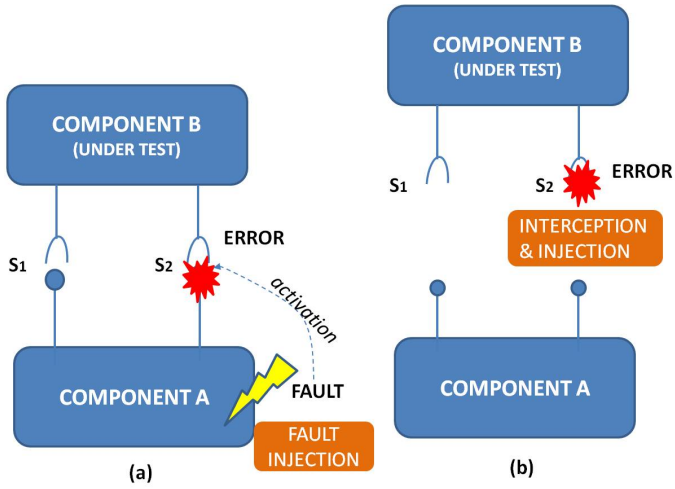

Figure 3.1: (a) Faults is injected into component A and after its activation becomes an error which emerges to the component interface. Through service S_2 the error propagates to component B. (b) The call to service S_2 is intercepted and an error is injected.

In both cases, the service interface is the error location. For this reason robustness testing is also termed *interface error injection* [39]. The driver interface and the application interface of the OS are typical target. These interfaces are of interests because through them is possible to assess the robustness of the OS against erroneous behavior of applications and drivers which have proven to be particulary error prone [40]. Nevertheless, it is possible to inject errors into other locations such as the hardware interface or into the interface of the internal components of the OS. A single experiment that consists in the injection of an error into the API of the component under test is also refereed as *test case*. The types of errors injected at the service interface exposed from the OS are classified in three categories [27]:

- **Fuzzy**, errors are chosen randomly among all possible values of the input domain of the service. Therefore experiments with this type of errors should be repeated a significant number of times to be confident in the final results.

- **Data Error**, errors are selected according to the type of the input

parameters. The selection of the error is conducted on the basis of the tester experience or with established methods (e.g., boundary analysis). As an example, since the interfaces of an OS are defined with C code, a data error for a parameter of type *int* is the MAX_INT value. Dependently on the type of the parameters, the number of injections can vary from one case to tens of cases for a given parameter [27].

- **Bit Flip**, errors are flip (permutation) of one of the bit of the input parameter of the service. This model derives from hardware errors in which the real faults are modeled as "bit flips" [1] [41]. It is easy to use, but it requires many experiments because of different number of the bit to flip for all the input parameters.

The errors can be injected in a precise temporal interval, **time-driven injection** or when specific events occur, **event-driven injection**. The time for the injection, often, is not precisely defined, rather it is assumed that the system is in a given state when executing the RT. The event-driven approach injects errors when a precise sequence of calls to the service interface takes place. The duration of the injection is assumed to be **transient** or **permanent**. An error originates from a fault, therefore its duration is related to it. Faults that turn into permanent are often considered as Bohrbugs 2.1 and they are detected easily with standard techniques [42]. Transient errors can be regarded as the manifestation of Mandelbugs or Heisenbugs (Sec.2.1) and are more likely to affect post-release software components. As a consequence, robustness testing injects transient errors.

3.2 Robustness Testing Applied to Operating Systems

Several studies approached the problem of robustness testing applied to operating systems. It is interesting to know that operating systems along with web applications are the COTS components with the larger number of applications of the RT [43]. One of the earliest study has been presented in [44], which evaluated the robustness of UNIX utilities in the presence of random inputs ("fuzzing"). Two tools, respectively *Fuzz* and *ptyjig*, were proposed to submit a random stream of characters to the target through

[1] More details about this fault moded are in Chapter 5

the standard input and through the terminal device. The study found that a significant number of utility programs on three UNIX systems (between 24% and 33%) is vulnerable to this type of errors, causing process crashes or stalls. A subsequent experiment [45] found that the same utilities were still sensible to a significant part of errors found in [44] 5 years later and that similar issues were present in network and graphical applications. These studies highlighted that robustness can be a serious concern even for mature, widely-adopted software. Moreover, the analysis pointed out several bugs, such as buffer overruns and unchecked return codes. Even if the fuzzing approach is simple to implement and can reveal robustness problems, its efficiency was questioned by some studies, since it relies on many trials and "good luck". In [46], it is pointed out that most of unstructured random tests only test the input parsing code of the program, and do not stress other software functions. *RIDDLE* [46], a tool for robustness testing of Windows NT utilities, extends fuzzing with erroneous inputs generated by a grammar that describes the format of inputs like a Backus-Naur form. These erroneous inputs (random and boundary values) are syntactically correct, and able to test more thoroughly the target program. In order to improve the efficiency of robustness testing, other studies investigated the *data-type based* error injection approach, which focuses on invalid inputs that tend to be more problematic than other to be handled. In [47], a data-type based approach is proposed to test a Real-Time OS (RTOS) kernel adopted in a fault-tolerant aerospace system. The RTOS is tested against invalid inputs passed to its *system call interface*, in order to assess the ability to handle errors generated by faulty user-space programs. The study considers system calls related to the file system (e.g., create, read, and write files), the memory system (e.g., allocation and deallocation of memory blocks), and the inter-process communication system (e.g., post a message and wait for a message). Each test consists of a system call invocation with a combination of both valid and invalid parameters. A test driver process executes the test case. For each group of system calls and each data type, the study defines a set of invalid input values (e.g., closed or read-only files, and NULL or wrong pointers to memory areas). The test outcome is determined by recording the error code returned by the system call (e.g., to identify whether the error code reflects or not the invalid input, or an error code is not returned at all), and by monitoring system processes using a watchdog process (e.g., a failure occurs if a process unexpectedly terminates during the experiment, or it is

stalled). The test campaign found several deficiencies in the target RTOS, some of them impacting seriously its reliability (e.g., a cold restart of the whole system is needed).

The approach of [47] was generalized in *BALLISTA* [26, 48], which was aimed at evaluating and benchmarking the robustness of commercial OSs with respect to the POSIX system call interface [49]. BALLISTA adopts a *data-type based* robustness testing approach, that is, it defines a subset of invalid values for every data type encompassed by the POSIX standard, and invokes system calls using several combinations of valid and invalid values.Examples of invalid inputs, using a data-type based approach on three data types, are provided in Table 3.1. The outcome of robustness test cases is classified by the severity of the OS failure, according to the *CRASH* scale:

- ***C**atastrophic*. The OS state becomes corrupted or the machine crashes and reboots.
- ***R**estart*. The OS never returns control to the caller of a system call, and the calling process is stalled and needs to be restarted.
- ***A**bort*. The OS terminates a process in an abnormal way.
- ***S**ilent*. The OS does not return an indication of an error in the presence of exceptional inputs.
- ***H**indering*. The OS returns an incorrect error code, i.e., the error code reports a misleading exceptional condition.

a Catastrophic failure occurs when the failure affects more than one task or the OS itself; Restart or Abort failures occur when the task launched by BALLISTA is killed by the OS or stalled; Silent or Hindering failures occur when the system call does not return an error code, or returns a wrong error code. BALLISTA found several invalid inputs not gracefully handled (Restarts and Aborts); a few Catastrophic failures were observed, mainly due to illegal pointer values, numeric overflows, and end-of-file overruns [26].

In [50–52], a *dependability benchmark* has been proposed to compare robustness of different OSs, by precisely defining the benchmark measures, the procedure and conditions under which the measures are obtained, and the domain in which these measures are considered valid and meaningful.In particular, to obtain realistic measures and allow a fair comparison, the OS is exercised using a *realistic* usage profile, which has be representative of the

Table 3.1: Examples of invalid input values for the three data types of the *write(int filedes, const void *buffer, size_t nbytes)* system call.

File descriptor (filedes)	**Memory buffer (buffer)**	**Size (nbytes)**
FD_CLOSED	BUF_SMALL_1	SIZE_1
FD_OPEN_READ	BUF_MED_PAGESIZE	SIZE_16
FD_OPEN_WRITE	BUF_LARGE_512MB	SIZE_PAGE
FD_DELETED	BUF_XLARGE_1GB	SIZE_PAGEx16
FD_NOEXIST	BUF_HUGE_2GB	SIZE_PAGEx16plus1
...	...	...

expected usage of the OS, since the test outcome is affected by *state* of the OS in which an invalid input occurs. The dependability benchmark defines realistic scenarios in which the OS is part of a database server system or mail server system, and the system is exercised using a representative set of user requests. System call inputs generated by user-space applications (e.g., the DBMS or the mail server processes) are intercepted and replaced with invalid ones, by using respectively data-type based values, random values, and bit-flips (i.e., a correct input is corrupted by inverting one bit). OSs are compared with respect to the severity of their failures, reaction time (i.e., mean time to respond to a system call in presence of faults) and restart time (i.e., mean time to restart the OS after a test). The dependability benchmark is one of the first effort to evaluate robustness tests while the target system is under different working conditions. Robustness testing of OSs has also been focused on device drivers, since they are usually provided by third party developers and represent a major cause of OS failures [40,53]. The robustness of the *Driver Programming Interface*, DPI, of OSs has been targeted in [54] and [55], in which invalid values are generated by faulty device drivers when they invoke a function of the OS kernel: in [54], invalid values are introduced using a data-type based approach, while in [55], the code of device drivers is mutated (by artificially inserting bugs) to cause a faulty behavior. Johansson et al. [56], and Winter et al. [57] later, compared the bit-flipping, fuzzing, and data-type based approaches with respect to their effectiveness in detecting vulnerabilities in the DPI of Windows CE, and the efforts required to setup and execute experiments. They found that

bit-flipping is the approach most effective at finding vulnerabilities, but it incurs a high execution cost due to the large number of experiments, thus providing a low efficiency, while the other approaches are more efficient but incur in a higher implementation cost (e.g., in the case of the data-type based approach, the user has to define exceptional values for each data type). Finally, they found that the best trade-off between effectiveness and cost is obtained by combining fuzzing with selective bit-flipping (i.e., focused on a subset of bits), since the two techniques tend to find different vulnerabilities. From all these studies, OSs result to be more vulnerable to device drivers than to applications, since developers tend to omit checks in the device driver interface to improve performance, and because they trust device drivers more than applications. Other works assessed the robustness of OSs with respect to hardware faults (e.g., CPU and disk faults), by corrupting OS code and data [58,59]. Similarly to system call testing, all these approaches either rely on a representative workload for exercising the system, or neglect the system state at all.

The influence of OS state gained attention in recent work on testing device drivers [60,61]. In [60], the concept of *call blocks* is introduced to model repeating subsequences of OS function calls made by device drivers, since they issue recurring sequences of function calls (e.g., when reading a large amount of data from a device): therefore, robustness testing is more efficient when it is focused on call blocks instead of injecting invalid inputs at random time. Sarbu et al. [61] proposed a state model for device driver testing, using a vector of boolean variables. Each variable represents an operation supported by the device driver: at a given time t, the i-th variable is true if the driver is performing the i-th operation. Case studies on Microsoft Windows OSs found that the test space can be reduced using the state model. Prabhakaran et al. [62] proposed an approach for testing journaling file systems, which injects disk faults at specific states of file system transactions. These studies showed that the OS state has an important role in testing such complex systems; however, they require knowledge about OS internals, and a manual analysis to define state models in which to inject faults.

3.3 Robustness Testing Applied to Other Software Systems

Robustness testing has been applied to a wide class of software systems. Regardless of the specific system, the key idea is to provide to its interface erroneous inputs or make the system experience exceptional condition. Groot [63] studies the behavior of a Knowledge Based System (KBS) in presence of "degraded inputs" (e.g., missing, incomplete and abnormal inputs). Malek [64] compares the robustness of Highly Available middlewares to exceptional inputs coming from the workload, the operating system and the hardware. Similarly, Kovi [65] tests the robustness of standard specifications-based HA middleware. Bovenzi [66] evaluates the robustness of data dissemination service compliant middleware. The exceptional inputs are selected as null and empty values, boundary values and values that can cause data type overflow.

Li [67] tests the robustness of a telecommunication systems, specifically, the error handling mechanism of the fault manager is tested against errors from the service manager, in the latter faults are deliberately injected (their approach resembles the one in 3.1(b)). Calori [68] presents a method to analyze the robustness of web applications based on FMEA and BBN. Fu [69] conceives a compile time analysis that allows to test error recovery code (i.e., exception handler) of Java web services. Barry [70] and Hanna [71] assess the robustness of web services with an approach based on analyzing the Web Service Description Language (WSDL) document of Web Services to identify what faults could affect the robustness attribute and then test cases were designed to detect those faults. Laranjero [72] provides an approach to automatically understand the behavior of the web server in presence of exceptional inputs, this saves a large amount of work since service responses has to be manually classified to distinguish regular responses from responses that indicate robustness problems.

Susskraut [73] proposes an automated approach to evaluate the robustness of software libraries, First, they use a static analysis to prepare and guide the following fault injection. In the dynamic analysis stage, fault injection experiments execute the library functions with both usual and extreme input values. The approach is experimented on Apache libraries. Zamli [74] takes advantage of the *reflection* to corrupt the input parameters of Java applications. Tarhini [75] presents a methodology for testing robustness of realtime

component-based systems using fault injection and adequate distributed test architecture. In this work, the term "hazard" refers to exceptional inputs and their insertion alters the expected event sequence. Vasan [76] and Jing [77] applies robustness testing to network protocol, here, the exceptional inputs are represented by faulty Packet Data Unit (PDU). Faulty PDUs are generated according to specific algorithms, which seed the field of the PDU with fault values. Notably, aside from a few works listed above [75, 77] they neglect if and how the state of the system under test influence the outcome.

Chapter 4

Stateful Robustness Testing of Operating Systems

Early studies on robustness testing have investigated in which way to conduct a robustness test, what type of faults/errors to inject and which layer of the operating system to target. However, the state of the system poses a vexing challenge to the use of this technique. The state of the systems is strictly linked to the instant in which to trigger the robustness test, thus influencing the final test outcome. In general, the state of an operating systems is complex to model because of its intricate design and implementation. We present two alternative approaches that extend the traditional robustness testing with the state of the target system, they paves the way for a stateful robustness testing. Stateful robustness testing shows the relevance of the state in robustness testing and outperforms conventional robustness testing in terms of both repeatability and number of experiments to conduct.

4.1 Approach I

4.1.1 Definitions

Since OS components can be very complex and their state has a significant influence on the OS correct behavior, it is necessary to take the states of the Component Under test (CUT) into account, and assess its robustness

as the state changes. According to this view, a hypothetical test plan is expressed through two dimensions: the exceptional inputs and the states. Inputs are selected as usual (e.g., boundary values) while the state varies in S = {s_1, s_2 ... s_n}. In order to apply this strategy, we need to test the CUT with both a test driver and a state setter. The former injects invalid inputs into its interface, whereas the latter is responsible for producing the state transition or keeping the component in a given state s_k. (see Figure 4.1) In complex components the state representation (i.e., the state model)

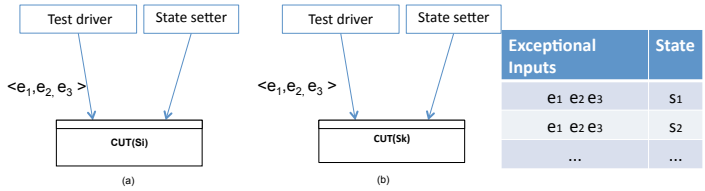

Figure 4.1: Robustness testing conducted with the CUT in two different states s_i and s_k.

plays a key role. It can be considered at several levels of abstraction, hence determining the number of potential states the state setter should cope with. This aspect is relevant for our approach, since it can affect the efficiency and the feasibility of robustness testing. Thus the state model should satisfy these requirements: i) it should be easy to set and control by the tester, ii) it should represent the state at a level of abstraction high enough to keep the number of test cases reasonably small and iii) it should include those configurations that are the most influential on the component behavior. Thus, with this regard, the model that we define expresses the state of an OS component without detailing its internals, since they are not always easy to understand and to manage, and would inflate the number of states.

4.1.2 Modeling the File System

In this work, we experiment the described strategy by applying it to the File System (FS) component. We choose the FS because it is a critical and bug-prone component [62, 78]. Furthermore, the behavior of the FS is influenced by its internal state and the other components with which it interacts (e.g., virtual memory manger, scheduler). Following the previous requirements, we conceived a model for the FS (4.2)

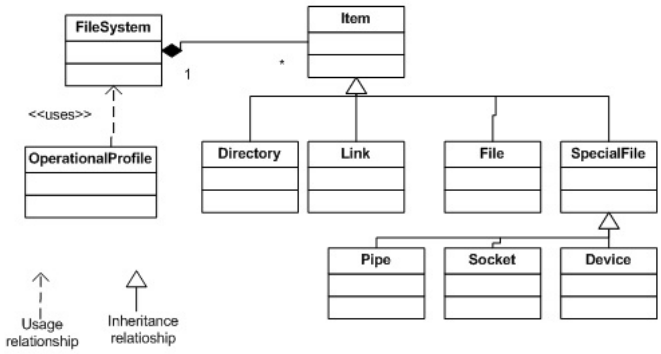

Figure 4.2: File System model.

Moreover, the model is easily adoptable across different FS [1] implementations; as a consequence, the proposed model does not take specific "internal design" of a FS into account (e.g., inode that are adopted in some UNIX file system, but not in others). The model is a UML representation of the FS, with three main classes: *Item*, *FileSystem* and *OperationalProfile*. *FileSystem* represents the contents of data on the disk as a whole. It includes the state attributes that are not specific of a file. The class attributes are reported in Table 4.1.

The choice of attribute values defines the test cases. Attributes like *Partition Allocated* can assume values from a minimum (e.g., 1MB) to the maximum allowable (e.g., 2TB). Therefore, the number of test cases, just for one parameter, grows rapidly. However, test cases in which the values of *Partition Allocated* varies with very small increments (e.g., from 1MB to 2MB) can be of little interest (e.g., 1MB or 2Mb both are values for a small partition). Thus, it is necessary to define criteria to keep the number of test cases reasonably low and cover a reasonable set of test scenarios. Hereafter, we illustrate potential choices for those attributes that the tester can set except for the attributes assigned by OS (e.g., *Max file size*). The attributes *Block size* and *Partition size* are typically set when the file system is formatted for the first time. In a hypothetical test campaign, these values could assume minimum, maximum and intermediate values. The attribute *Partition*

[1] In this work, the term "File System" refers to the OS component for managing files. The term "filesystem" refers to the contents on the storage, e.g., the structure of tree.

Table 4.1: FileSystem attributes.

Attributes	**Description**	**Type**
Partition type	Typology of the partition	Primary, Logical
Partition size	Size of the partition on which is installed the FS	Byte
Partition allocated	The current size of the allocated partition	Byte
Max file size	The maximum dimension of a file on the FS	Byte
Block size	The dimension of a block	Byte
FS implementation	The type of file system	NTFS, ext2, ext3
# of files allocated	The number of files in the FS	Integer
# of directories	The number of directories in the FS	Integer
FS layout	The tree that represents the FS	Balanced, Unbalanced
# of items allocated	The current number of items allocated in the FS	Integer

allocated can be expressed as a percentage of *Partition size*, therefore the tester can set scenarios in which the file system is totally full, partially full or empty. The attribute *FS layout* deals with the tree representing the directory hierarchy on the FS. In particular, it can assume the values: balanced, i.e., trees in which the number of sub-directories is almost the same on each directory, and unbalanced, i.e., trees in which the number of sub-directories significantly differs. In order to generate balanced and unbalanced trees, we introduce $P(\{d_{k+1}d_j\})$, i.e., the probability that a new directory, d_{k+1}, is a child of a directory, d_j, already present in the tree. This probability allows, to some extent, to control the structure of the hierarchy, once *Number of Directory allocated* is fixed. For $P(\{d_{k+1}d_j\})$, we provide the following formulas for generating balanced and unbalanced trees, although other choices

are possible (e.g., to use a well-known statistical distribution):

$$P_{unbalanced}(\{d_{k+1}d_j\}) = depth(d_j)\frac{1}{\sum_{i=1}^{k} depth(d_i)} \tag{4.1}$$

$$P_{balanced}(\{d_{k+1}d_j\}) = \frac{1}{depth(d_j)}\frac{1}{\sum_{i=1}^{k} \frac{1}{depth(d_i)}} \tag{4.2}$$

$$ParentDirectory = d : max\{P(\{d_{k+1}d_1\})...P(\{d_{k+1}d_k\})\} \tag{4.3}$$

where k is the number of current directories in the tree, and N the number of directories to be created; k is increased until k=N. In 4.1, new directories are more likely to form an unbalanced tree, since the higher the depth of a node is, the higher the probability to have children. In 4.2, new directories are more likely to group at the same depth. The parent directory 4.3 is the one with the highest value of $P(\{d_{k+1}d_j\})$. As for the *FileSystem* class, it is possible to conceive several criteria for assigning values to the attributes. For instance, the attribute *Name* can assume alphabetical and numerical characters with equal probability or the length should not overpass a given value. The attributes *Permission* and *Owner* can be assigned in such a way that a given percentage of files are executable by the owner only, another percentage is readable by all users and so on. The attribute *Size* can be fixed for all files, generated according to a statistical distribution.

The *Item* class represents the entity which a *FileSystem* is made of. For this class, we define typical attributes that are available in every OS. Such attributes are: *name of the item*, *permission* (e.g., readable, writeable, executable), *owner* (root, nobody, user) and *size*. The classes that inherit from *Item* represent the different types of file in a UNIX file system. Files are randomly generated to populate the directory tree mentioned above; the location and type of file can be determined according to statistical distributions. The FS, like other OS subcomponents, uses resources such as cache, locks and buffers. We refer to these resources as auxiliary resources, that is, resources that serve for managing an *Item* of a FS. For instance, if a thread performs I/O operations it is likely to stimulate auxiliary resources: indeed, buffers are instantiated; locks to control the access to them are used, and so forth. These resources are part of the internal state of the FS, although they are not included in our model, since (i) they cannot be easily controlled by

the tester, and (ii) they are dependent on the FS internals. Moreover, most of these resources are instantiated at run-time, and they are not part of the filesystem on the disk. The presence of these resources, however, cannot be neglected because they may influence the state of the FS and potentially change test outcomes. Therefore, in order to include both the behavior of the auxiliary resources in our model and the manner in which the FS is exercised, we introduce the *OperationalProfile* class. It expresses the degree of usage of the auxiliary resources and more generally, the way the FS is stimulated. This class does not directly model the auxiliary resource, but it allows to know the way in which the FS is invoked while performing a test. Thus the tester, indirectly, is aware of the mechanisms that are stimulated, e.g., if there are threads invoking I/O operations it is likely that caching and mutex mechanisms are invoked. The *OperationalProfile* attributes are reported in Table 4.2.

Table 4.2: *OperationalProfile* attributes

Attributes	**Description**	**Type**
Number of tasks invoking FS ops.	Number of tasks that invokes I/O operations (like read, write, open).	Integer
Average number of ops/s	Average number of operations made by a task	Integer
Ratio of read/write ops.	Ratio of read/write operations made by a task	Float

The *OperationalProfile* attributes are related to the performance of the *File System* and the hardware, which can limit the rate of FS operations that can be served by the system within a reasonable latency. Therefore, the selection of these attributes should be preceded by a capacity test aiming at assessing the maximum operation rate allowed by the system. A capacity test [79] consists in gradually increasing the operations rate, given a fixed number of concurrent tasks (e.g., 2, 4 or 16), until the I/O bandwidth is saturated, i.e., the amount of transferred data per second reaches its peak. After that the I/O bandwidth is known, the tests can select a discrete set of usage levels (e.g., 10% and 90% of I/O bandwidth) and the ratio between read and write operations (e.g., 2 read operations per 1 write operation).

4.2 Approach II

4.2.1 Definitions

In this alternative approach, named StAte-Based Robustness testIng of operatiNg systEms (SABRINE), we distinguish between the different *components* that form an OS. A component is a subsystem of the OS that is responsible for managing a resource or for providing a set of services, such as memory management, I/O management, and process scheduling. Each component provides an *interface* to other components, that is, a set of functions that are invoked to request a service. Applications can require a service to the OS by performing a *system call*, which in turn triggers one or more components that interact in order to implement the OS service (Figure 4.3). Additionally, component services can be invoked by *interrupt requests* coming from the hardware, and by *kernel tasks*, i.e., processes that execute in kernel space and that can directly interact with OS components. Several system calls, interrupt requests and kernel tasks can be executed in parallel (by alternating on the same CPU, or by running concurrently on different CPUs). The applications that run on top of the OS and exercise it are referred to as the *workload*.

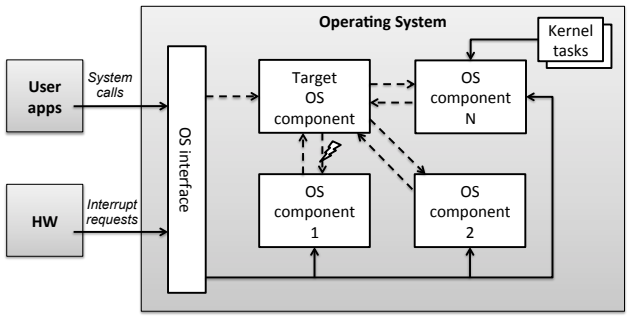

Figure 4.3: System overview.

We test OS robustness against *service failures* of a component. A service can fail, for instance, due to the exhaustion of a resource, or due to a hardware fault in a device or a defect in an OS component. In case of a failure, the function that has been invoked typically returns an *error code* to notify that a service cannot be provided. A service failure may cause a *non-robust*

behavior, such as an OS crash, when it is not correctly handled by the OS code that invokes the service. In such a case, the OS is considered *vulnerable* to that service failure, and a *robustness vulnerability* has been found, which may require to fix the OS in order to make it robust against the service failure (e.g., by retrying the failed operation, or switching to a degraded mode of service). To test robustness, we force a service failure (also referred to as *fault*) while the system is exercised with a workload, that is, by forcing the called function (representing the service) to return an error code, and analyzing the system reaction to the service failure.

In particular, given that the same service can be requested by several OS components, we focus on service invocations performed by one specific *target component* at a time. For instance, the target component can be represented by a new component under development, such as a device driver to support new hardware, or a new filesystem component. To identify the states of the target OS component, we log *interactions* at its interfaces with other OS components (dashed arrows in Figure 4.3). The target component may be invoked by another component (*input interaction*), or the target component may invoke another component (*output interaction*). An interaction with a function that can fail (e.g., a function for resource allocation), and in which a failure can be injected, is referred to as *injectable interaction* (see Figure 4.4). We both consider the case in which an injectable interaction is *direct*, that is, the injectable function is invoked by the target component, and the case in which the injectable interaction is *indirect*, in which the injectable function is invoked by another component on behalf of the target component (i.e., the function is invoked to provide a service to the target component). We include indirect interactions in our robustness tests since the fault may propagate to the target component and trigger its robustness vulnerabilities.

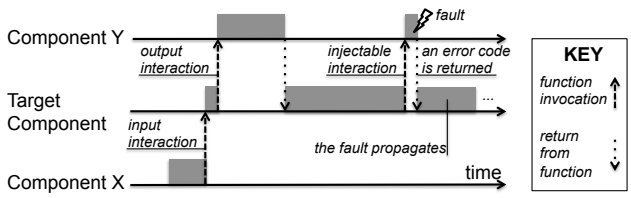

Figure 4.4: Interactions among OS components.

4.2.2 Behavioral Data Collection

In this phase, the OS is executed using a workload, without injecting faults. The workload is selected by developers and testers before performing robustness testing. In a similar way to performance benchmarks, the workload reflects the context in which the OS will be adopted (e.g., web applications, DBMSs, ...), and it affects the way the OS is exercised during the tests (e.g., a DBMS-oriented workload stresses storage-related services) and its behavior under unexpected events. During the workload execution, we monitor component interactions (Figure 4.3), and derive state models for the target component on the basis of its interactions with other components. As discussed later, state models are based on *input*, *output*, and *injectable interactions* that involve the target component.

Component interactions are monitored through *static* (i.e., hard-wired in the kernel source code) or *dynamic probes* (i.e., inserted at run-time) located at the interfaces of components. A probe consists of a small piece of code (e.g., a breakpoint) that is inserted in a given code location, and that triggers a *handler routine* when executed. In turn, the handler collects information and restores kernel execution. For tracing input interactions, we probe the interface of the target component, storing information (as described in the next section) about the component that invokes the target. For tracing output and injectable interactions, we probe the interfaces of components that are invoked by the target component, storing information about the invoked component. Probing at component interfaces represents a practical solution for most of modern commodity OSs, since they often provide tools that allow to insert static and/or dynamic probes in kernel code and monitor its execution with a low overhead, such as DTrace for Oracle Solaris [80], SystemTap for Linux [81], and DebugView for Microsoft Windows [82]. Data from probes are transferred to an external computer through a *serial port*, which is typically adopted for debugging purposes since the serial port driver has minimal interactions with other OS components, thus limiting interferences on OS execution due to monitoring.

When collecting behavioral data, we need to account for the fact that component interactions may vary between different workload executions due to random factors: for instance, some interactions may appear in a different order or not appear at all, depending on the timing of I/O events and process scheduling. As a consequence, such random factors can affect the definition of robustness test cases, since some OS states can be missed during an indi-

vidual workload execution. For this reason, we repeat the execution of the workload several times during this phase: by doing so, we are able to include sets of interactions even when they do not occur at every execution, and to generate robustness test cases that also cover them, leaving uncovered only the few sets of interactions that occur very rarely.

4.2.3 Pattern Identification

The output of the previous phase consists of an *interaction log*, in which interactions among components appear in sequential order (i.e., ordered by their timestamp). The log is divided into *sequences*, where each sequence is a set of events that occur during the execution of an individual system call, interrupt request, or a kernel task. Two executions of the same system call represent two distinct sequences, but they can produce different sets of interactions, depending on the state of the system. By dividing the log into sequences, we discriminate subsets of interactions that repeat identically (in this phase) or are similar (in the next phase), in which it is likely that the target has assumed the same states. Identical sequences are grouped together, forming a *pattern*.

To extract sequences, we log the following information for each interaction between the target and other components:

- **Operation ID**: A string identifier of the operation (system call, interrupt request, or kernel task) that is being serviced at the time of the interaction.
- **Execution ID**: An integer that identifies a specific execution of a system call, interrupt request or kernel task. Each time that an operation starts, a new execution ID is generated, and all the interactions produced during this operation will be identified by this value. If a system call is started while the same system call is already executing (e.g., invoked by a different process on a different CPU), the interactions of the new execution have the new execution ID, while the interactions generated by the other operation are still denoted by the previous execution ID.
- **Trace ID**: An integer that identifies a specific execution of the whole workload. Since the workload can be executed several times, and more than one workload execution can appear in the same interaction log,

the interactions of each workload execution are denoted by a specific trace IDs.

Moreover, each logged interaction contains the following information:

- **Called function**: In the case of input interactions, it is the name of the function of the target invoked by another component. For output and injectable interactions, it is the name of the function invoked by the target in another component.
- **Call point**: The code location in which the function is invoked (e.g., the address in the executable code of the instruction that invokes the function).

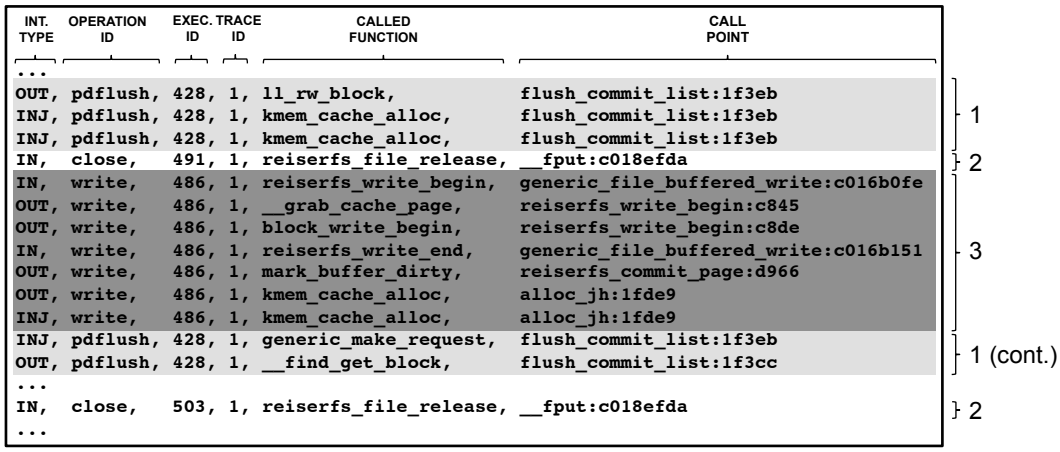

Figure 4.5: Example of interaction log and pattern identification.

Figure 4.5 shows an extract of the interaction log from the case study that we will consider in this work. The first sequence in the example (highlighted in light gray) is identified by the triple $\langle pdflush, 428, 1\rangle$, in which there are two output interactions (denoted by "OUT") and two injectable interactions (denoted by "INJ"), and all of them are invocations made by the *flush_commit_list* function of the target component (a filesystem). This sequence is interleaved with two other ones, identified by $\langle close, 491, 1\rangle$ (white background) and $\langle write, 486, 1\rangle$ (dark gray background) respectively. This interleaving occurred since *pdflush* (a kernel task) has been suspended while

executing *kmem_cache_alloc*, which performs memory allocation and can preempt a task when this operation takes a long time (e.g., an I/O operation is required in order to free memory). The other two sequences are generated by the workload invoking the *close* and *write* system calls, which in turn trigger input interactions with the target component (denoted by "IN"). When the third sequence performs its second memory allocation, a workload process is preempted in favor of *pdflush*, which continues the first sequence. The same sequence of interactions can repeat identically in the log, with a different identifier: this is the case of the sequence identified by $\langle close, 503, 1\rangle$ (a sequence containing only one interaction), which is identical to the sequence identified by $\langle close, 491, 1\rangle$. These sequences represent two instances of the same pattern (number 2), and only one instance per pattern is considered in the subsequent phases.

4.2.4 Pattern Clustering

The execution of the OS typically leads to patterns that are not identical, but differ for a few interactions, or there is a small variation in the order of the interactions. In other words, several patterns tend to be very "similar". Small variations in the sequences are unavoidable, and are due to non-deterministic factors that can affect OS execution. For instance, Figure 4.6 shows two similar patterns related to the *write* system call (for the sake of readability, only the called function is showed for each interaction). The patterns p_1 and p_2 exhibit almost the same number and sequence of interactions, aside from three interactions (gray background) which appear only in p_2. These interactions, in this specific case, represent the allocation of additional memory when metadata are written to the disk.

However, generating one behavioral model for each individual pattern would lead to an excessive number of models and, as discussed later, to superfluous robustness test cases. Therefore, before generating behavioral models, we group together similar patterns, thus obtaining *clusters* of patterns. Each cluster represents a specific "mode of operation" of the target component, where the patterns in a given cluster only differ with respect to a few interactions. To perform clustering, we first measure the similarity among all pairs of patterns using a *similarity function*, and then we cluster patterns that are similar with a *clustering algorithm*.

A similarity function is a quantitative way to express the similarity between two sequences, and it is used in several applications, such as the pro-

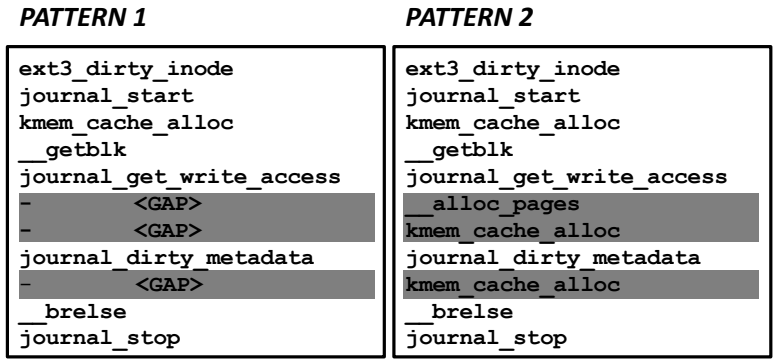

Figure 4.6: Example of similar patterns.

cessing of biological sequences. In our case, we compare sequences of interactions, in which each interaction (i.e., a pair $\langle$ *called function*, *call point* $\rangle$) represents an element of the sequence. Two main approaches exists in the literature for evaluating similarity, which respectively (i) only consider which elements appear in each sequence, and evaluate the number of elements that appear in both sequences (*set-based similarity functions*), and (ii) consider the ordering of elements while comparing common elements between the sequences (*sequence-based similarity functions*) [83]. In our approach, we measure the similarity between patterns with a sequence-based function: two sequences of interactions with different orderings may reflect different states of the system, and should be regarded as dissimilar.

Sequence-based functions are based on "alignment" algorithms, in which the elements of the sequences are placed side by side in order to maximize the number of matches, and minimizing the number of gaps and mismatches[2]. With the Smith-Waterman algorithm [84], we compute an alignment score for each pair of patterns p and q according to the following dynamic programming formulation:

$$F_{0,j} = -j * g \quad , \quad F_{i,0} = -i * g \tag{4.4}$$

[2]When the elements at a given position of a pair of patterns are the same, we say that there is a *match*; otherwise we say that there is a *mismatch*. A *gap*, instead, consists in introducing a special symbol to fill the vacuum due to the different lengths of the two sequences.

$$F_{i,j} = max \begin{cases} F_{i-1,j-1} + sim(p_i, q_j) \\ F_{i-1,j} - g \\ F_{i,j-1} - g \\ 0 \end{cases} \tag{4.5}$$

$$sim(p_i, q_j) = \begin{cases} m & \text{if } p_i = q_j \\ -n & otherwise \end{cases} \tag{4.6}$$

In this set of equations, p_i and q_j are the i-th and j-th element of patterns p and q, respectively, with $i \in [1, ..., N]$ and $j \in [1, ..., M]$, and N and M are the lengths of patterns p and q. F is the scoring matrix, where the value $F_{i,j}$ is the score of the best alignment between the initial segment $p_{1...i}$ of p up to p_i and the initial segment $q_{1...j}$ of q up to q_j, which is calculated recursively from $F_{i-1,j}$, $F_{i,j-1}$, and $F_{i-1,j-1}$ [84]. The constants g and n are the score penalty for gaps and mismatches, while m is a score reward for matches. Common choices are $g = 1$, $n = 2$, and $m = g + n$ [83]. The highest value of F, that is $F_{N,M}$, represents the score of the best possible alignment. For instance, the patterns p_1 and p_2 showed (aligned) in Figure 4.6 have score $SW(p_1, p_2) = W_{match} * m + W_{mismatch} * n + W_{gap} * g = (8) * (+3) + (0) * (-2) + (3) * (-1) = 21$, where the Ws are the number of matches, mismatches and gaps, respectively. For each pair of patterns, we compute the SW score, and collect this score into a cell of a matrix, named Similarity Matrix (SM), which expresses quantitatively the degree of similarity among all pairs of patterns. The score of each pair is normalized using the length of the longest pattern in each pair, since patterns in our context have variable length, which would otherwise affect the evaluation of pattern similarity.

We group together similar patterns using a *spectral clustering* algorithm [85]. This class of algorithms allows to cluster a set of elements on the basis of their similarity matrix, and has recently emerged as an effective and computationally-efficient clustering approach [86]. A spectral clustering algorithm interprets input elements as the nodes of a graph, and the similarity score of each pair of elements as the weight of the connection between two nodes. Then, elements are clustered into k groups, by performing k cuts in the graph, each group includes the nodes that are still connected after the cuts. The idea behind spectral clustering is that cutting "weak" connections splits the graph into partitions of elements that are "strongly connected" and thus very similar each other. The weights of cuts in the graphs are closely

related to the *spectrum* of the graph, that is, the eigenvalues $\lambda_1, \ldots, \lambda_n$ of the laplacian matrix L derived from SM [87]. By processing L on the basis of its eigenvectors, the spectral clustering algorithm can obtain k cuts and, in turn, k clusters. To select the number k of clusters, we use the *eigengap* heuristic [86], which chooses k such that all $\lambda_1, ..., \lambda_k$ eigenvalues of L are very small and λ_{k+1} is relatively large. Intuitively, if the first k eigenvalues are very small, then the algorithm can split the graph into k parts without separating strongly-connected nodes.

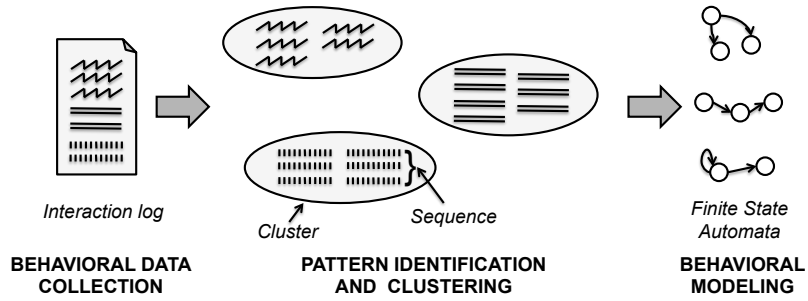

Figure 4.7: Summary of the model generation approach.

4.2.5 Behavioral Modeling and Test Suite Generation

At this point, the initial interaction log, through pattern identification and clustering, has been turned into clusters of sequences. From each cluster, we infer a behavioral model in the form of a *finite state automata* (FSA). Behavioral models of software systems have been adopted and proved to be useful in several software engineering applications, such as specification mining [88], automated debugging [89], and reverse engineering [90]. We adopt the kBehavior mining algorithm [91, 92], which incrementally infers FSAs from execution traces, which in our case consist of sequences of component interactions. The algorithm starts with an empty automata (e.g., only one state with no transitions), examines the first pattern and generates an FSA whose transitions are labeled with an interaction (i.e., a pair $\langle$ *called function*, *call point* $\rangle$). If the cluster contains more than one pattern, they are subsequently provided as input to the mining algorithm, one at at a time. Each time that a new pattern is provided, the algorithm augments the FSA with new transitions and states, in order to reflect both the new patterns

and previous ones. This process is repeated for each cluster, leading to an FSA for each cluster. The overall transformation of the behavioral data, from the interaction log up to the FSAs is depicted in Figure 4.7. Finally, a set of robustness test cases is derived from each FSA. We identify transitions in the FSA that represent an injectable interaction (i.e., an invocation of a function that can fail), and introduce a test case for each injectable interaction in the FSA. Given a state S with an outgoing transition t that represents an injectable interaction, the test case associated with t consists in forcing a failure of that function when the system is in the state S and the injectable function is invoked.

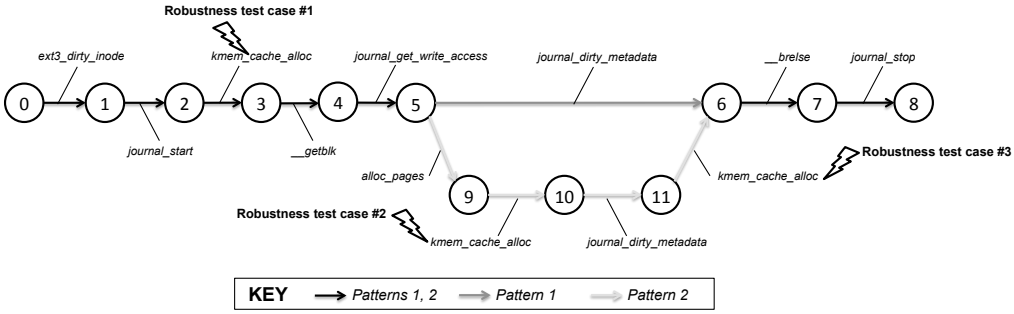

Figure 4.8: Example of behavioral model.

Figure 4.8 provides an example of behavioral model. It depicts the FSA obtained from the two patterns in Figure 4.6 (only the called function is showed at each transition). States from 0 to 5, and states from 6 to 8 are connected by interactions that appears in both patterns, while states 5 and 6 are connected by two different sets of transitions. This occurs since the two patterns share most of their interactions, but one of them performs contains additional memory allocations, and the mining algorithm inserted new states and transitions corresponding to these interactions. Assuming to inject failures at the invocations of the *kmem_cache_alloc* memory allocation function, this FSA leads to 3 robustness test cases. The example also points out the importance of clustering on the generation of test cases. The first invocation of *kmem_cache_alloc*, which appears in both patterns of Figure 4.6, is performed in the same context in both patterns. By using only one FSA for representing both patterns, the occurrences of the first *kmem_cache_alloc* invocation are represented by only one transition in the FSA, the one be-

tween states 2 and 3. In this way, we can reduce the number of robustness test cases (only one test is performed for each transition with an injectable interaction), while still covering relevant states of the target system, thus improving the efficiency of robustness testing. A similar reduction is obtained when the *kmem_cache_alloc* is performed in a loop: in such cases, since the same interactions are repeated several times, the mining algorithm translated these interactions into a loop in the FSA, and only one test case is generated for the injectable interaction in the loop.

4.2.6 Test Execution

Robustness test cases are translated in *test programs* that are then executed to inject service failures in the different states of the OS. In a similar way to the "Behavioral Data Collection" phase (Subsection 4.2.2), the test program collects interaction sequences at run-time using kernel probes, and keeps track of the current state of the target component. If the test program, in the current state, observes the interaction specified in the FSA, it transits to the new state. If the observed interaction it is not the expected one, the target program transit to the initial state. When the system reaches the target state S, the target program injects a service failure during the injectable interaction. After the injection, the OS behavior evolves freely.

4.3 Case Study

To illustrate the use of both approaches (Approach I and Approach II), we consider an OS developed in the context of a pilot R&D project, in conjunction with the Finmeccanica s.p.a. industrial group. The goal of the project is to develop a reliable Linux-based Real-Time Operating System (RTOS), namely FIN.X-RTOS to adopt in software systems for avionic applications. FINX-RTOS 2.2.1 derives from the Plain-Vanilla 2.6.24 kernel, it is reduced to an essential subset of components (e.g, there is only the code for a specific board) and it is extended with real-time services (e.g, the interrupts are threaded and real-time mutexs replace the ordinary mutexs). In order to certify FIN.X-RTOS, the OS needs to be accompanied by evidences (e.g., test artifacts) showing the compliancy to the recommendations of the DO-178B safety standard [5]. Therefore, the entire kernel code has been documented and accompanied with low-level and high-level requirements. The

requirements of the standard at level D (to be followed for software whose anomalous behavior would cause "a minor failure condition" for the aircraft) have been fulfilled. At the time of writing, FIN.X-RTOS is being tested with additional verification activities according to the requirements of level C (for software that may cause "a major failure condition" for the aircraft), which demand to test the robustness of the software against abnormal inputs and conditions. An example of requirement from the standard is to "*provoke transitions that are not allowed by the software requirements*" [5].

4.4 Approach I: Experimentation

The proposed approach has been applied to the ext3 file system available in FIN.X-RTOS. We selected a set of system calls to test, described in Table 4.3. The system calls are commonly used by applications and exercise different parts of the FS code.

Table 4.3: System calls tested.

System Call	Description
access	check user's permissions for a file
dup2	duplicate a file descriptor
lseek	reposition read/write file offset
mkfifo	make a FIFO special file (a named pipe)
mmap	map files or devices into memory
open	open and possibly create a file or device
read	read from a file descriptor
unlink	delete a name and possibly the file it refers to
write	write to a file descriptor

To apply the proposed strategy, we selected, without loss of generality, two well known tools for supporting testing execution, namely Ballista and Filebench [3]. With regard to Figure 4.1, Ballista plays the role of test driver, while FileBench is the state setter. The Ballista tool is currently distributed with the Linux Test Project tool suite. We ported the original version to

[3]http://www.ece.cmu.edu/ koopman/ballista/ - http://www.fsl.cs.sunysb.edu/ vass/-filebench/

FIN.X-RTOS. FileBench is a tool for FS benchmarking: the user can customize a workload by configuring I/O access patterns in terms of number of threads, access type and so on. In our test campaign, we choose a realistic scenario in which the partition of filesystem is partially full (75% of Partiton size) and there are tasks invoking FS operations, e.g., read and write. Leveraging on the model introduced in Section 4.1.2, we create a logical partition with a balanced tree and the number of directories is 10 each one populated with 100 small files. No other items have been considered. Table 4.4 summarizes the values that we selected for the *FileSystem* entity's attributes. Table 4.5 shows the values selected for the *File* entities; all the files, apart from *Name*, have the same values. Table 4.6 specifies the attributes of *OperationalProfile*, which are typical values for FS benchmarking.

Table 4.4: *FileSystem* values.

Attribute	**Value**
Partition type	Logical
Partition size	2GB
Partition allocated	1,5GB
Block size	4096
File system implementation	ext3
Number of files allocated	1000
Number of directories allocated	10
Number of items allocated	1010

Table 4.5: *File* values.

Attribute	**Value**
Name	Numeric string with length equals to five
Permission	Readable, Writeable, Executable
Owner	Root
Size	1500Kb

Those instances of File, *FileSystem*, and *OperationalProfile* reproduce stressful conditions in which to test the FS. By stressing the FS with read

Table 4.6: *OperationalProfile* values

Attributes	Values
Number of tasks invoking FS operations	16
Average number of operations per second	10
Ratio of read/write operations	1

and write operations on a full allocated partition, we aim at creating exceptional conditions: in fact, with this setting, it is more likely to experiment conditions in which disk blocks are not available, seek operations have to traverse several directories, and so on. We carry out three experimental campaigns:

1. Stateless robustness testing. Ballista injects errors into the selected system calls (Table 4.3). The errors to apply to the parameters of the system call belongs to the default Ballista configuration. An example is represented in Section 3.1(Table 3.1). This test campaign lasts 15 minutes.

2. Stress testing. FileBench invokes the system calls read and write on the files previously allocated for 1 hour. The operations produced by FileBench reflect the attributes of *OperationalProfile* (Table 4.6). Ballista is not executed.

3. Stateful robustness testing. FileBench and Ballista work at the same time. Ballista and FileBench use the same configuration (error model and operations executed) of the previous campaigns. The entire test campaign lasts 1 hour. The experimental duration for the first test campaign is the time that Ballista spends to execute all the test cases. The second campaign lasts the time necessary for Ballista to execute all the tests while FileBench is running. The time for the third test campaign is set to 1 hour in order to compare the results between the second and third campaign over the same duration time.

4.4.1 Results

We first analyze the outcomes of robustness tests, which are classified according to the CRASH scale: a Catastrophic failure occurs when the failure

affects more than one task or the OS itself; Restart or Abort failures occur when the task launched by BALLISTA is killed by the OS or stalled; Silent or Hindering failures occur when the system call does not return an error code, or returns a wrong error code (for more details see Section 3.2).

Table 4.7 provides the summary produced by Ballista in the default configuration (i.e., all potential test cases are generated). We did not observe any Catastrophic failure, and only a small number of Restart and Abort failures occurred. This result was expected, since the OS is a mature and well-tested system, and is consistent with past results on POSIX OSs [26], in which only a small number of corner cases led to Catastrophic failures (e.g., an OS crash). The relevance of Restart and Abort failures is a controversial subject, since OS developers tend to consider them as a "robust" behavior of the OS [26]. According to this point of view, we do not consider Restarts as severe failures: several OSs (e.g., QNX, Minix) intentionally deal with a misbehaving task by killing it in some specific cases (e.g., manipulation of an invalid memory address, or lack of privileges for performing an operation), in order to avoid further error propagation within the system. Similarly, Abort failures can represent an expected (and desirable) behavior of the OS, such as in the case of the *read()* and *write()* system calls that can bring a task in a "waiting for I/O" state. For these reasons, a "Restart" or "Abort" outcome cannot be considered as a "failure" without a detailed analysis of the expected behavior. It should be noted that stateful robustness testing differs from stateless robustness testing with respect to the number of Restart outcomes, mostly due to failed memory and disk allocations. Although we cannot conclude that these outcomes represent OS failures, this result points out that OS state can affect test outcomes and the assessment of OS robustness.

However, the stateful tests cover a scenario not considered by stateless tests, and therefore they represent an additional evidence of the robust behavior of the OS. As a result, we observed an increased coverage of kernel code after executing the stateful tests; this aspect is relevant since coverage is a measure of test confidence and a requirement for software in safety-critical systems (e.g., DO-178B at level C [5]). We analyzed statement coverage of file system code, which is the target of our tests. The file system code is arranged in three directories: the code in the "fs/" directory is independent from the specific file system implementation (i.e., it is shared among several implementations such as ext3 and NTFS). The "ext3" directory pro-

Table 4.7: Results of robustness tests.

Function	**# Tests**	**Stateless RT**		**Stateful RT**	
		# Restart	# Abort	# Restart	# Abort
access()	3,986	0	4	1	4
dup2()	3,954	0	0	1	0
lseek()	3,977	0	0	0	0
mkfifo()	3,870	0	5	1	5
mmap()	4,003	0	0	0	0
open()	3,988	0	8	40	8
read()	3,924	0	253	1	253
unlink()	500	0	1	0	1
write()	3,989	0	68	4	68
Total	32,191	0	339	48	339

vides the implementation of the ext3 file system; finally, the "jbd" directory provides a generic support for journaling file systems. Data about coverage was collected using GCOV. Table 4.8 compares the statement coverage with respect to the three considered scenarios. We observed differences in coverage between stateless (second column) and stateful robustness testing (fourth column), ranging between 0.49% and 15.11%. Part of the code is covered by the plain state setter (i.e., without using Ballista); the remaining part is covered due to interactions between Ballista and the OS state (some examples are provided in the following).

In particular, stateful testing exercised those parts of the file system that interact with other subsystems (e.g., interactions between "fs/buffer.c" and the memory management subsystem, and between "fs/fs-writeback.c" and disk device drivers). The coverage improvement is more significant for the journal-related code (i.e., the JBD component in "fs/jbd"). This effect can be attributed to the interactions between file system transactions and the state of I/O queues. For instance, a transaction commit can be delayed due to concurrent I/O operations, therefore affecting the management of data buffers within the kernel and the file system image on the disk. Although the improvement is less significant for the implementation-independent code, the proposed approach has been useful for improving test coverage with no

human effort. This aspect is relevant since FIN.X-RTOS is mostly composed by third-party code re-used from the Linux kernel; covering this code can be very costly, due to the lack of knowledge of kernel internals and the inherent complexity of OS code (e.g., heuristics for memory management).

Table 4.8: Statement coverage.

Source file	**Stateless robustness testing**	**Stress testing**	**Stateful robustness testing**
fs/binfmt_elf.c	319/850 (37.53%)	331/850 (38.94%)	332/850 (39.06%)
fs/buffer.c	529/1320 (40.08%)	553/1320 (41.89%)	565/1320 (42.80%)
fs/dcache.c	371/880 (42.16%)	341/880 (38.75%)	387/880 (43.98%)
fs/exec.c	479/807 (59.36%)	392/807 (48.57%)	486/807 (60.22%)
fs/fswriteback.c	146/273 (53.48%)	169/273 (61.90%)	174/273 (63.74%)
fs/inode.c	252/527 (47.82%)	307/527 (58.25%)	316/527 (59.96%)
fs/namei.c	918/1392 (65.95%)	626/1392 (44.97%)	925/1392 (66.45%)
fs/select.c	237/402 (58.96%)	237/402 (58.96%)	239/402 (59.45%)
fs/ext3/balloc.c	384/556 (69.06%)	385/556 (69.24%)	398/556 (71.58%)
fs/ext3/dir.c	140/219 (63.93%)	143/219 (65.30%)	144/219 (65.75%)
fs/ext3/ialloc.c	181/337 (53.71%)	186/337 (55.19%)	189/337 (56.08%)
fs/ext3/inode.c	719/1204 (59.72%)	729/1204 (60.55%)	737/1204 (61.21%)
fs/ext3/namei.c	607/1088 (55.79%)	654/1088 (60.11%)	781/1088 (71.78%)
fs/jbd/checkpoint.c	102/263 (38.78%)	141/263 (53.61%)	142/263 (53.99%)
fs/jbd/commit.c	300/362 (82.87%)	302/362 (83.43%)	318/362 (87.85%)
fs/jbd/revoke.c	108/228 (47.37%)	105/228 (46.05%)	116/228 (50.87%)
fs/jbd/transaction.c	489/697 (70.16%)	500/697 (71.74%)	545/697 (78.19%)

In order to better understand the interactions between OS state and test cases, we analyzed more in depth part of the kernel code only covered by stateful robustness testing. Figure 4.9 shows an example of corner case in the kernel code not covered in stateless testing (part of the code was omitted; we kept some comments from developers). The *real_lookup()* routine is invoked when file metadata are not in the page cache, and the FS needs to access to the disk. It blocks the current task on a semaphore (using the *mutex_lock()* primitive) until a given directory can be accessed in mutual exclusion. It then checks if metadata have been added to the cache during this wait period. Usually, metadata are not found, and the routine performs an access to the disk. In stateful testing, a different behavior was observed, since the

cache has been re-populated during the wait period (developers refer to this situation as "nasty case"), and additional operations are executed (e.g., to check that metadata are not expired due to a timeout in distributed file systems). This code was only executed in stateful testing due to interactions with the cache that occur when concurrent I/O operations are taking place.

```
1     static struct dentry * real_lookup(struct dentry * parent,
2     struct qstr * name, struct nameidata *nd) {
3     /* --- OMISSIS (declarations) --- */
4     mutex_lock(&dir->i_mutex);
5     result = d_lookup(parent, name);
6     if (!result) {
7         /* --- OMISSIS (performs lookup) --- */
8         mutex_unlock(&dir->i_mutex);
9         return result;
10    }
11    /* Uhhuh! Nasty case: the cache was re-populated while
12    we waited on the semaphore. Need to revalidate.*/
13    mutex_unlock(&dir->i_mutex);
14    if (result->d_op && result->d_op->d_revalidate) {
15        result = do_revalidate(result, nd);
16        if (!result)
17            result = ERR_PTR(-ENOENT);
18    }
19    return result;
20 }
```

Figure 4.9: Example of kernel code covered due to interactions between the file system and caching (from *real_lookup()*, fs/namei.c:478).

Another example is provided in Figure 4.10, which is related to concurrency of kernel code. The *ll_rw_block()* routine performs several low-level accesses to the disk, and each access is controlled by a "buffer head" data structure. During the inspection of the list of buffer heads, one of them could have been locked by another concurrent task; this condition is detected by the *test_set_buffer_locked()* primitive, which may fail to lock the buffer head in some cases. Stateful testing covered this rare scenario, and it is worth being tested to verify that pending I/O is correctly managed.

Finally, we analyzed an example of kernel code interacting with memory management, which is provided in Figure 4.11. The *try_to_free_buffers()* routine is invoked by the file system when the cache for file system data (the "page cache") gets large and pages need to be freed for incoming data. It may occur that a file system transaction involves I/O buffers allocated over several pages, and these pages cannot be de-allocated until the transaction

```
1     void ll_rw_block(int rw, int nr, struct buffer_head *bhs[]) {
2     int i;
3     for (i = 0; i < nr; i++) {
4         struct buffer_head *bh = bhs[i];
5         if (rw == SWRITE)
6             lock_buffer(bh);
7         else if (test_set_buffer_locked(bh))
8             continue;
9         /* ---- OMISSIS (performs I/O op.) ---- */
10 }
```

Figure 4.10: Example of kernel code covered due to concurrent I/O requests (from *ll_rw_block()*, fs/buffer.c:2941).

commits. Pages are then marked with "mapping == NULL" in order to be reclaimed later (the *drop_buffers()* routine checks that I/O buffers in the page are not being used). As suggested by the comment in the code, this condition is unlikely to occur; the code has been executed in stateful testing since memory management has been put under stress.

```
1     int try_to_free_buffers(struct page *page) {
2     /* ---- OMISSIS (declarations) ---- */
3     BUG_ON(!PageLocked(page));
4     if (PageWriteback(page))
5         return 0;
6     if (mapping == NULL) {      /* can this still happen? */
7         ret = drop_buffers(page, &buffers_to_free);
8         goto out;
9     }
10    /* ---- OMISSIS (page writeback and deallocation) ---- */
11 }
```

Figure 4.11: Example of kernel code covered due interactions between the file system and memory management (from *try_to_free_buffers()*, fs/buffer.c:3057).

4.5 Approach II: Experimentation

We applied the SABRINE approach to assess the robustness of a set of I/O-related components against service failures in the memory allocator of the kernel. We selected memory allocation failures because most of OS

components depend on this service, and it is a frequent cause of system failures [93]. Kernel developers also perceive memory allocation problems as a likely cause of OS failures: in fact, the Linux kernel includes a framework for injecting service failures, which encompasses memory allocation failures [94]. Both in our implementation of SABRINE and in the Linux injection framework, a failure is injected by forcing a memory allocation function (*kmem_cache_alloc*) to return a NULL pointer instead of a valid pointer to the newly allocated memory.

While the Linux framework injects failures at a random time, the SABRINE approach selects the time in which to inject based on the state of the target component. We compare both these approaches in our experiments. The relationships between I/O-related components in the kernel are showed in Figure 4.12. In this architecture, I/O system calls (e.g., *write*s) first pass through the Virtual File System, which provides generic services for implementing file systems, and forwards a file operation to the specific filesystem that manages the file (e.g., EXT3, ReiserFS, ...). The file system can issue an I/O operation to the Block I/O Layer, which provides generic services such as scheduling of I/O requests and caching of disk data. In turn, the Block I/O Layer forwards requests to a device driver, which manages the disk device.

All these components use the memory allocator for dynamically allocate memory, such as for storing file metadata and for temporary I/O buffers. The target components are represented by thick boxes in Figure 4.12, and include two widely-adopted file systems (EXT3 and ReiserFS) and a device driver (the SCSI subsystem). We adopt the *Apache HTTPD* web server to exercise the OS, using the *httperf* performance testing tool to generate web requests [95].

Experiments were executed in a virtual machine environment, and were fully automated using programs running on the host machine. A SystemTap program [81] collects behavioral data. FSA models are created with the kBehavior algorithm [91, 92], and automatically translated in test programs implemented in the SystemTap language. Behavioral data and error messages from the OS are collected using virtual serial port connections. In the case of an OS crash, we collect information including the type of exception (e.g., illegal memory access), the code location, the contents of the stack and of CPU registers. The virtual machine is automatically rebooted in case of a crash.

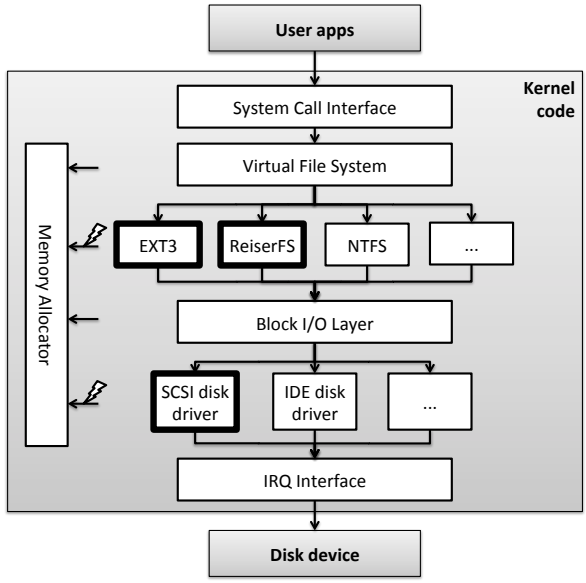

Figure 4.12: Overview of I/O-related subsystems in FIN.X-RTOS.

4.5.1 Results

This section presents the results obtained from SABRINE in our case study. Table 4.9 provides some basic facts about data collection and processing, and test generation. We collected an interaction log for each target component, by running the web server workload 10 times for each target. In the cases of EXT3 and ReiserFS, interactions were performed in the context of file-related system calls, such as *open* and *write*, and of the *pdflush* kernel thread of the Block I/O Layer, which periodically flushed cached data to the disk. For SCSI, interactions were initiated by kernel threads of the Block I/O Layer, which requested data transfers, and by interrupts from the hardware.

Several thousands of interactions appear in each log. These logs were divided into sequences, and identical sequences were grouped into patterns. Since we aimed at injecting service failures of the *kmem_cache_alloc* function, we focused our analysis only on sequences containing an interaction with this function. For EXT3 and ReiserFS, a non-negligible number of patterns and clusters was generated, since memory allocations were performed in many different contexts during filesystem operations. Instead, even if

Table 4.9: Statistics on the behavioral data collection and test case generation.

	EXT3	**ReiserFS**	**SCSI**
# interactions	34,784	97,341	27,311
*# sequences**	432	239	1,307
*# patterns**	79	57	10
# clusters	9	6	2
# test cases	49	28	10

** involving the* kmem_cache_alloc *function.*

SCSI produced the highest number of sequences with *kmem_cache_alloc*, it exhibited the lowest number of distinct patterns and of clusters: for this target, memory allocations performed always at the same code location (i.e., when allocating memory for storing a new data transfer command), leading to repetitive sequences. Consequently, only 2 clusters are enough to group the patterns of SCSI, while EXT3 and ReiserFS require 9 and 6 clusters, respectively.

We examined in depth the clusters, in order to understand the mode of operation represented by each pattern, and to assess whether clustered patterns are "semantically" similar. Table 4.10 provides a description for the clusters of EXT3; similar interpretations apply to ReiserFS and SCSI clusters, but we do not show them due to space constraints. Column "Behavior" provides a brief description of clusters, and column "Context" details the system calls or kernel task in which these behaviors were observed. Each cluster represents a distinct behavior of the file system. For instance, cluster 1 gathers the patterns representing "get" and "set" operations on file metadata (e.g., file permissions), which is the case of the *stat* system call; clusters 2 and 3 represent the typical behavior of *read* and *write* system calls. For each cluster of each target component, we derived an FSA, and a set of one or more test cases for each FSA (Subsection 4.2.5).

In order to evaluate the efficiency of SABRINE, we executed the robustness test cases generated by the approach, and compared the results with the ones obtained using the standard fault injection framework included in the kernel [94]. In the standard injection framework, allocation failures are

Table 4.10: Clusters for EXT3.

Cluster	**Behavior**	**Context**	**# patterns**
1	gets and sets the file metadata	*stat* syscall	6
2	retrieves and stores in memory the file index block, or updates it on the disk	*open*, *unlink* syscalls	5
3	copies file contents from disk to a cache, and modifies it	*write* syscall	8
4	copies a small amount of data from a file to a network socket	*sendfile* syscall	10
5	modifies the contents of a file already in the disk cache	*write* syscall	8
6	flushes a small amount of data from the cache to the disk	*pdflush* kernel task	19
7	flushes a large amount of data from the cache to the disk	*pdflush* kernel task	6
8	copies a large amount of data from a file to a network socket	*sendfile* syscall	12
9	updates file metadata to reflect that is has been memory-mapped	*mmap2* syscall	5

injected randomly: each time *kmem_cache_alloc* is invoked, it can fail with a fixed probability P; if a failure is not injected, the subsequent invocation becomes the next candidate injection. Moreover, the standard injector allows to inject service failures when the injectable function is invoked (directly or indirectly) by the target component (EXT3, ReiserFS, or SCSI) [94]. We performed 1,000 random injections for each target component; these experiments took a few days per target component to complete, therefore 1,000 injections can be considered a conservative estimate on the number of experiments that a developer would perform. We set $P = 5\%$ in order to avoid that too many injections take place only at the beginning of the experiment. As for SABRINE, we executed the number of tests reported in the last row of Table 4.9. We classify the outcome of a test in:

- **Kernel Failure**: the OS is crashed, or its state is corrupted. To detect state corruptions in the OS, we enabled several consistency checks introduced by developers in the kernel code, including checks on stack overflows, stuck system calls, locks not released, and corruptions on key kernel data structures. This kind of failures is the most severe, since they affect all applications and the OS itself.
- **Workload Failure**: the web server crashes, exits abnormally, does not reply to requests, or does not execute correctly the requests. These failures are detected through the logs of the web server and of the client.
- **FS Corruption**: after each test, we detect disk corruptions using filesystem check utilities.
- **No Impact**: neither the OS nor the workload show an abnormal behavior.

Table 4.11 summarizes the percentage of failures observed during the experiments. Both kernel failures and workload failures were observed; instead, no memory allocation failure caused filesystem corruptions, since the kernel tends to crash immediately or to fail gracefully in order to avoid data corruptions. The SCSI target component was very robust to memory allocation failures: by inspecting its source code, we found that it keeps a pool of previously-allocated data structures (e.g., data transfer command structure) that supply memory when the kernel allocator fails, in order not to lose important disk writes. Instead, there were several cases in which an injected

Table 4.11: Statistics on failure distributions.

Testing Technique	Target	Kernel Failures	Workload Failures	FS Corruptions
Random	EXT3	32.8%	37.5%	0%
	ReiserFS	9.6%	65.9%	0%
	SCSI	0%	0%	0%
SABRINE	EXT3	22.4%	16.3%	0%
	ReiserFS	20.0%	32.0%	0%
	SCSI	0%	0%	0%

failure in EXT3 and ReiserFS lead workload and kernel failures: in the first case, the injection caused a system call failure and, in turn, a failure of the web server; in the second case, the kernel performs an illegal memory access, leading to an OS crash.

Frame no.	*Kernel function*
0	`kmem_cache_alloc+0x22/0x110` ← **a failure occurs here**
1	`radix_tree_node_alloc+0x35/0xb0`
2	`radix_tree_insert+0x16e/0x1d0`
3	`add_to_page_cache+0x65/0x1d0`
4	`add_to_page_cache_lru+0x1b/0x40`
5	`mpage_readpages+0x70/0xe0`
6	`ext3_readpages+0x19/0x20` ← **affected EXT3 function**
7	`__do_page_cache_readahead+0x176/0x210`
8	`ondemand_readahead+0xbe/0x170`
9	`page_cache_async_readahead+0x66/0x90`
10	`generic_file_splice_read+0x4a9/0x630`
11	`do_splice_to+0x61/0x80`
12	`splice_direct_to_actor+0x8f/0x180`
13	`do_splice_direct+0x3b/0x60`
14	`do_sendfile+0x187/0x240`
15	`sys_sendfile64+0x77/0xa0`
16	`sysenter_past_esp+0x5f/0x91`

Figure 4.13: Call stack of a robustness vulnerability.

In particular, OS crashes were caused by two robustness vulnerabilities in the kernel code. For instance, Figure 4.13 shows the case of a memory allocation in *radix_tree_node_alloc* that causes the corruption of data struc-

Table 4.12: Percentage of random injection tests that trigger each vulnerability.

Vulnerability	**EXT3**	**ReiserFS**
__get_blk	29.0%	0.2%
radix_tree_node_alloc	3.8%	9.4%

tures when the allocation fails and, in turn, the failure of the OS component calling the function. This vulnerability emerges when the file system is retrieving data from the disk to its cache in the main memory when a memory allocation fails, as in the case of cluster 3 in Table 4.10. Table 4.12 provides the percentage of random tests able to reveal each robustness vulnerability: this percentage can be very low, as the case of *__get_blk* in ReiserFS (two cases out of 1,000 random tests trigger the vulnerability).

In our experiments, SABRINE was able to detect both the two vulnerabilities, with a high efficiency. For each vulnerability, SABRINE generated several test cases able to detect it, by injecting in states where the vulnerability could be triggered. The SABRINE approach identified the same vulnerabilities of random testing, but only a relatively small set of robustness test cases was required to find them (77 test cases in total). Moreover, a vulnerability can be easily reproduced once a test case of SABRINE can detect it. By repeating 10 times the execution of SABRINE test cases, almost every OS crashes repeated identically: Table 4.13 provides the average probability of repeating an OS crash. Instead, it is difficult to reproduce failures using random injections, since the state of the system at the time of the injection plays an important role in triggering vulnerabilities, but it is neglected in random injections. The dramatic reduction of the number of test cases and the ability to easily reproduce OS failures increase significantly the efficiency of robustness testing.

Table 4.13: Probability to reproduce a robustness vulnerability in SABRINE.

Vulnerability	**EXT3**	**ReiserFS**
__get_blk	68.8%	100%
radix_tree_node_alloc	77.7%	100%

Chapter 5

Techniques for Injecting Hardware Faults

From the nineties several techniques have been developed for the assessment of robustness against hardware faults. Early methods relied on additional hardware devices that had some limitations such as the low controllability and repeatability of the experiments. As the software started to be a viable alternative for emulating hardware faults, a number of tools have been developed under the umbrella approach Software Implemented Fault Injection (SWIFI). SWIFI is attractive because does not require purposely developed device, thus saving cost, and allows to assess the robustness or fault tolerance easier. In this chapter, we survey the techniques used for emulating hardware faults and highlight the robustness of software systems against them.

5.1 Introduction

Early studies in the fault injection field evaluated the robustness of software component when injecting or emulating hardware errors. These studied assumed that the hardware is faulty and its behavior can impact the executing software program, the workload. It must be said that the term error injection and fault injection are sometimes used interchangeably. However, in this work we will be consistent with the definition in Section 2.1. We inject faults into the hardware layer to induce a failure. The hardware failure prop-

agates to the software layer as an error and we observe the robustness of the software to them. Nowadays, the reason for the injection of hardware faults reside in the continuous scale of the transistor to small dimensions. Thereby, the transistor become more susceptible to transient faults, also called soft errors or Single Event Upset (SEU), mainly due to the following factors:

- *radiation*, atmospheric neutrons result from cosmic rays colliding with participles in the atmosphere. Neutrons with energies greater than 1 mega-electron-volt (MeV) when strike a sensitive region of an SRAM cell, the charge that accumulates could exceed the minimum charge that is needed to keep the value stored in the cell, resulting in a soft error [96].

- *crosstalk*, which is the capacitive and inductive coupling of signals from one signal line to another. As system performance and board densities increase, so does the problem of cross-talk [97].

- *wear-out effects*, under this term are collected critical intrinsic failure mechanisms for processors such as electromigration, stress migration, gate-oxide breakdown or time dependent dielectric breakdown (TDDB), and thermal cycling [98]. These effects can result in soft errors or even in hard errors (a persistent error).

5.2 Hardware Implemented Fault Injection

Hardware implemented fault injection (HIFI) includes additional hardware to inoculate faults/errors in the target hardware. This technique can be applied with and without contact to the target on the basis of the location in which to inject. HIFI techniques requiring contact with the target are *pin-level fault injection* and *test port-based fault injection* while for *radiation based fault injection* is not necessary the contact.

5.2.1 Pin-level Fault Injection

There are two approaches to implement this technique: the *forcing* and the *insertion*. In the forcing approach the faults/errors are injected through probes to the pin of the Integrated Circuit (IC). AFIT [99] supports this technique. In the insertion approach, the IC is isolated from the system with a specific design circuit that intersects the signals to the IC. RIFLE

[100] implements this approach, while MESSALINE [101] either forcing or insertion.

5.2.2 Test Port-Based Fault Injection

This technique takes benefit of test ports available on modern microprocessor. JTAG [1], NEXUS and Background Debug Mode (BDM) are three common test port types that equip several microprocessors, the first two are standardized by IEEE, while the latter is a proprietary solution by Freeescale Inc. Test ports access instruction set architecture registers (ISA) as it is the case for JTAG and also memory word (BDM and Nexus).The injection through these ports involves three major steps: i) a breakpoint is set before executing the workload, ii) when the workload reaches the breakpoint the value in the target location (e.g., a register) is corrupted iii) the execution is resumed. Therefore, these techniques allow to control the experiment with respect to the target location, but the execution time depends on the access speed of the test ports (Nexus in general is faster than JTAG and BDM). This technique is implemented in GOOFI [102], INERTE [103], FlexiFi [104].

5.2.3 Radiation-Based Fault Injection

An injection consists in exposing the target to Electromagnetic Interferences (EMI) or to Heavy-Ion radiation [105]. This technique has been used in the nineties [106] and early twenties, the main drawbacks are the low controllability in terms of fault location (the interference insists on different area of the board) and its repeatability.

5.2.4 Power Supply Disturbance

The fault injection occurs with a voltage drop in the power supply of the processor for a few milliseconds [105, 107]. When the power voltage drops below a predefined threshold is expected that the processor does not work properly. This injection is hard to perform especially in modern microprocessor because their very high clock frequency make calculation of the pulse intensity tricky. Therefore, the intrusiveness and repeatability of this injection is questionable.

[1] Joint Test Action Group is the common name for the IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture.

5.3 Software Implemented Fault Injection

Software Implemented Fault Injection (SWIFI) emulates hardware faults through the software layer. Basically, there are two main SWIFI techniques: *pre run-time injection* and *run-time injection*. Pre run-time injection analyzes the binary image of the workload before its execution and alter it in a specific fault location. This techniques is also refereed as *Instrumentation Based Fault Injection* [108] and can be implemented as follow. The fault injector inserts a software breakpoint in the workload executable. In Figure 5.1 the instruction 3980_{16} is replaced with a jump to the injection routine at the address 7000_{16}. The injection routine manipulates a register, executes the replaced instruction 3980_{16} and resumes the execution of the workload.

INJECTION STEP	ADDR	ASSEMBLY	
0	397c	add	r9,r2,r1
1	3980	~~lwz~~	~~ro,8(r9)~~
2		ba	7000 <instr_f>
5	3984	cmp	r0,r9
3	inst_f:	#instrumentation	function
	7000	stw	r1,-16(r1)
	...	...	...
	7048	lwz	ro,8(r9)
4	704c	ba	3984

Figure 5.1: Instrumentation of the workload.

This technique has a variable spatial and temporal intrusiveness because it extends the original image with injection code of variable size. The run-time injection encompasses five major steps. The injector downloads on the target an exception routine designed to inject faults. The second step sets an hardware breakpoint along with an invocation count which serve as a fault trigger. Third, the workload is executed. Fourth when the workload triggers the hardware breakpoint, the hardware exception code inject the fault. Fifth, the workload resumes.This technique does not manipulate the binary of the workload, but it implements a complex mechanisms that can increase its temporal intrusiveness.

Pre-injection and run-time injection have been intensely used over the

years and experimentally compared in [109], on the average they produce the same results. SWIFI assumes that hardware faults can be emulated with one of the two techniques, however faults manifesting in the internal logic of the Central Processing Unit (CPU) are not reproducible (e.g., faults in the Arithmetic Logic Unit). Yet, permanent hardware faults are hard to emulate with SWIFI because their insertion implies several manipulation of the target or of the workload. On the opposite, SWIFI can effectively inject transient faults.

A novel approach for SWIFI benefits of the *Extensible Firmware Interface*(EFI) [110] standard available on x86/x64 architecture. In principal, this approach does not modify the target and is highly flexible with four possible implementations, however, at present, there is no fault injection campaign that proves the feasibility of this technique.

5.4 SWIFI: approaches and tools

We describe in this section the evolution of SWIFI over the years and the tools which implement it. FIAT [31] developed in 1990 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 (1992) [16] 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 emulates hardware and software faults [2] on the kernel of Sun OS 4.1.2. FINE (1993) [111] can inject transient and permanent hardware faults in the CPU, bus and memory (text and data area) by flipping a bit. DEFINE (1994) [20] 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 (1995) [112]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

[2]For details about software faults see Section 3.1

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 (1996) [113] performs injection on TANDEM system and supports 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 (1998) [114] 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 (1999) [115] 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 [22] 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. Exhaustif [116] (2007) adds the target workload with a software module that can manipulate memory and processor registers according to specific patterns, such as changes in the state of a bit (bit flip), the use of a mask (bit mask) or copy of a new value. Skarin [17] extends the previous version of Goofi [102] to inject multiple bit flip and bit flip into CPU registers and memory. Goofi-2 supports 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.

5.5 Robustness of Software to Hardware Faults

In this section, we survey the literature with the aim to highlight the sensitivity of software components to hardware errors injected through SWIFI. The analysis includes contributions concerning single node systems (e.g., a microprocessor with a workload), since the contribution of this thesis is restricted to this kind of systems. Hence studies regarding distributed systems are left out. Barton [31] injects 130000 faults into an IBM system on which execute two workloads, quicksort and matrix multiplication. Both quicksort

and matrix multiplication run with inputs of different data size and they are not extended with software fault tolerance mechanism (e.g., checksum), neither an operating system is present. Error detection mechanism are provided by the hardware. Experimental evidences highlighted that there is a strong linear correlation between the coverage of detection mechanism and data size, the first decreases as the second increases.

Kanawati [16] injected faults and errors into a Sparc system under the execution of three workloads: matrix multiplications using checksum, quicksort with assertions, and matrix multiplication using Continuous Signature Monitoring (CSM) and checksum. The workloads were targeted with 8 different fault models. Results for the matrix multiplication show that about 43% of 600000 injections are caught by Sparc error detection systems (e.g., segmentation fault) and workload built-in detection mechanisms (e.g., checksum). While over 41% of the injected transient errors were latent (No Error). Results for quicksort indicate that failure distribution varies when a 100 elements array and 1000 elements are provided, in particular when the data size increase, the probability of corrupting a data element increases as well.

Kao [111] investigates the robustness of a Unix OS to hardware and software faults. In this case the workload are synthetically generated to invokes OS system calls. Results illustrate that the fault locations (data segment, text segment, bus) induce fluctuation in the failure distribution: the percentage of faults provoking a self reboot of the OS change from 0.76% (faults in bus) to 0.22% (faults in data segment). This is explained because a significant part of the UNIX kernel is not exercised, although the accelerated workload is used.

Tsai [117] benchmarks two different implementation of Tandem system, Tandem A and Tandem B. Three workloads are synthetically generated, one is CPU intensive, another is I/O intensive and the last one is a balanced mix of CPU, I/O and memory intensive operations. Faults are injected into both memory and CPU registers according to single bit flip, multiple bit flip, and zero/set model. The three workloads on the two target systems show a different error sensitiveness. In Tandem A the mixed workload is more robust to errors while in Tandem B the CPU bound workload is the more robust. Apparently, only the I/O workload is the less robust across the two different implementation of Tandem.

Carreira [114] performs fault injection on a PowerPc board, the fault model is bit flip. Three workloads, Π-calculation (computes an approximate

value of π), SOR (a Laplace equation solver), and matrix multiplication are targeted with about 2000 faults each. The iterative nature of the Laplace equation solver algorithm masked a high percentage of the faults and therefore they obtained a small number of silent data corruption (no detection mechanism caught the fault and the elaboration differs from the expected one), not more than 8.5% in all functional units (e.g. address bus, floating point unit). The matrix multiplication enhanced with the ABFT (Algorithm Based Fault Tolerance) mechanism allows to detect all the errors. ABFT is an extremely simple method, which only implies the inclusion of an extra line and column in the result matrix.

Benso [115] carries on fault injection campaigns on a Motorola board. Parser (a syntactical analyzer for arithmetic expressions with a software error detection mechanism), matrix multiplication (of two matrixes, 10x10 elements each), a bubble sort algorithm (running on a 10 integers vector) are submitted to fault injection campaigns of approximately 300000 faults in CPU registers and memory. In their experience, the size of data structure affects the percentage of "no-impact" faults, because larger data size entails the injection of faults into variables or registers outside the period in which it is used. Bubble Sort and Parser are control-dominated programs therefore faults are likely to trigger a detection mechanism or have no effect. Conversely, matrix is data-dominated, hence faults are more prone to generate a silent data corruption.

Audet [118] shows, through synthetic programs and a real program that implements the Fast Fourier Transform algorithm (FFT), how the error sensitivity is affected from the program structure. The synthetic workloads implement the same functionality, but some are a pure sequence of instructions, while others are a mix of iterative instructions (*for* loop) and sequential instructions. Blocks of iterative instructions seem much more robust to transient errors than the equivalent sequential implementation. This consideration is also backed up by the FFT workload. From this experimental observation, Audet deducts a set of rules to apply during coding to make the workload more robust. Examples of rules are: i) "If data dependency is not affected, break a large loop into a series of smaller loops" ii) "iterative computations (loops) should be placed toward the end of a program whenever possible".

Folkesson [119] estimated the percentage of value failure for quicksort and shellsort, both executed with 24 different inputs on a Thor board. Quicksort

has a percentage of value failure varying from 8% up to 14% whilst for shellsort the percentage of value failure is approximately constant and equals to 18%.

Ruiz [24] benchmarks the robustness of an engine control application running directly on a PowerPc board and when it executes on the same board but with the interposition of an operating system. The engine control application drives the quantity of the air and fuel to enter the cylinder. Representative inputs recorded from a real engine stimulates the control application. A campaign consisting of 2000 transient faults reveals that 10.7% of injections produces an unpredictable behavior of the stand alone engine control application (without operating system) while unpredictable situation is reduced by a factor of four when the application executes on the operating system.

In the last years, the need for more dependable software in area where the cost is a major issues has encouraged researchers in developing software fault tolerant algorithms and mechanisms, the so called Software Implemented Hardware Fault Tolerance (SIHFT). The software component is then hardened with additional source or assembly code. Oh [120] with its seminal work presents the *Error Detection by Data Diversity and Duplicated Instructions* (ED^4). This technique executes two "different" programs with the same functionality but with different data sets and compare their outputs. Although, the effectiveness of ED^4 against single event upset is proved only through simulation, this work has been inspiring for many other studies conducted on real systems.

Madeira [121] is among the first authors to propose a software detection schema based on signature checking. The assembly is segmented in section. A signature is computed for each section and used at run time to detect errors. This approach is applied to: 1) a pseudo-random number generator, 2) a string search program, 3) a bit shift, set, reset, and test program, 4) a quick sort, and 5) a Sieve prime number generator. Unfortunately, results about the robustness of each application are not available, however the proposed detection mechanism is able to detect about 94% of 6000 faults injected in the Z80 bus.

Rebaundengo [122,123] introduces a set of rules to make more robust the software to transient errors. Such rules are the duplication and checksum of variables, control flow checks, and they have been applied to the source code of: Sieve program which implements the sieve of Eratosthenes, matrix

multiplication and bubble sort running on Intel 8085. The proposed method can reduce to zero the number of silent data corruption regardless of the fault location (data and code).

Nicolescu [124] provides the results of fault injection into Leon processor registers and a digital signal processing unit for 6 workloads of which 3 synthetic programs and 3 real programs. The three synthetic programs are: intensive data computing program (IDC), intensive branching program (IB) and the high recursivity degree program (HRD). The three real workloads are Constant Modulus Algorithm (CMA), signal equalization algorithm used in space communications, quick sort, and Finite Impulse Response (FIR) a digital filter. All workloads are extended with advanced control flow techniques and code duplications. An extensive fault injection on both target hardware show the absence of silent data corruption for all the workloads.

Reis [125] proposes software fault tolerance mechanisms implemented at assembly level. The mechanism duplicates code instructions and check the control flow with checksum. These methods are combined with hardware implemented error correction code. Transient faults are injected into CPU registers accessed by a large number of programs from SPEC CINT2000, SPEC FP2000, SPEC CINT95 and MediaBench benchmark, in total more than 32 programs are targeted while executing on Intel Itanium. Results show that for all the program no silent data corruption is observed.

Vinter [126] augments the robustness of a software implemented integrator with a schema that protects the internal state of the integrator itself. The proposed solution proved to be very effective against single and multiple bit flip injected into the registers of a PI controller: no value failure is observed.

Skarin [127] tested the robustness of anti-lock brake by wire system (ABS) against bit flip in registers and memory area of a PowerPc board. The ABS is augmented with two error detection mechanisms: a stack pointer checker and a rate limit check, an ad-hoc schema to check the output values of an integrator. Together these mechanisms can reduce critical failures (an unacceptable behavior of the ABS) from 4.6% to 0.4%.

Cook [128]investigates on the masking effects at assembly instruction level, i.e., for a class of assembly instructions (e.g., load, store, compare) he shows how transient faults do not alter the behavior of the workload. The masking effects are a sort of "workload self-immunity" to transient errors. The experimentation is conducted on SPEC CPU2000 INT benchmarks compiled with different optimizations. Interestingly, across all the optimiza-

tions and all the programs about 30% of 10000 injections is masked. Cooks exploits this masking effects to develop more efficient hardware detection mechanisms.

Alexandersson [129] benchmarks a Fibonacci program and an Anti-lock Brake by wire System (ABS) with fault tolerance mechanisms implemented either through Aspect Oriented Programming (AOP) or manually. The fault tolerance mechanisms are an improved version of the traditional control flow check (CFC) and the triple time redundant with forward recovery (TTR). Workloads run on a PowerPc board. The total value failure obtained when using the CFC mechanism varied between 12% and 5% for the evaluated programs. Value failure amounts from 4% to 6% for the TTR mechanism. These results are valid regardless of the implementation method, AOP or manual addition of C lines of code. Another notably contribution by Alexanderson is the evaluation of the influence of the compiler optimization on the failure distribution. Aggressive optimization are likely to reduce the size of the program thus lowering the exposure to faults, as a consequence highly optimized version of the two workloads presented a smaller percentage of value failure.

From this literature review emerges that, at present, there is no common agreement on the workloads to use in fault injection campaigns, thus the results, here reported, are not meant for comparison. Yet, only the work [16, 24, 31, 119] provide insight on the influence of the workload inputs on the failure distribution. Since inputs can impact on the results to a large extent, they must be contemplated in fault injection experiments. Surprisingly, many of the analyzed studies do not even state which input exercises the workloads.

Chapter 6

An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions

Despite the consolidated use of SWIFI, there are still open problems which deserve an investigation such as the influence of the workload inputs on the experimentation. Thus, we experimentally illustrate the influence of the workload inputs on fault injection outcomes and study the relationship existing between them. This chapter presents the results of extensive fault injection experiments with four programs where single bit errors were injected in CPU registers and main memory locations of the target systems. The aim of the study is to investigate how error coverage varies for different inputs. We conducted experiments with programs protected by triple-time redundant execution with forward recovery, and with programs without software-implemented fault tolerance. In addition, we propose a technique for identifying input sets that are likely to cause the measured error coverage to vary.

6.1 Target Workloads

We present the four target workloads used in the fault injection setups; secure hash algorithm (SHA), cyclic redundancy check (CRC), quick sort (Qsort), and binary string to integer convertor (BinInt). SHA is a cryptographic hash function which generates a 160-bit message digest. Here we use SHA-1 algorithm which is applied in many security protocols and applications such as SSL, TLS, SSH and IPsec. The CRC that we used in our experiments is a software implementation of CRC 32-bit polynomial which is mostly used to calculate the end-to-end checksum. Qsort is a recursive implementation of the well-known quick sort algorithm, which is also used as a target program for fault injection experiments in [119, 130]. Finally, BinInt converts an ASCII binary string, 1s and 0s, into its equivalent integer value.

Even though the implementation of our workloads can be found in the MiBench suite [1], we only take CRC and BinInt from this suite. For the quick sort algorithm, the MiBench implementation uses a built-in C function named qsort whose source code is not available. This prevents us from performing detailed analysis. Furthermore, the MiBench implementation of SHA uses dynamic memory allocation which is not necessary for an embedded system.

Thus, we adopt another implementation of SHA [2]. The structure of these synthetic workloads profoundly differs in terms of lines of source code (LOC), number of functions, input types and executed assembly instructions. BinInt is the smallest workload with 7 LOC and is made of one function with one loop, whereas SHA measures 125 LOC and has 5 functions.

6.1.1 Input Sets

Nine different inputs are selected for each workload. The combination of an input and a workload is called an *execution flow*. Thus, for each workload, we have conducted experiments for 9 execution flows. On the basis of the length of the inputs, we group SHA and CRC execution flows into three categories, see Table 6.1 and Table 6.2. These categories are chosen to represent input lengths that are common in real scenarios. For Qsort, the input vector consists of 6 integers. The execution flows use the same 6 integers with different permutations (Table 6.3). The input of BinInt is a random string

[1] Mibench Version 1, [Online] http://www.eecs.umich.edu/mibench/
[2] http://www.dil.univ-mrs.fr/ morin/DIL/tp-crypto/sha1-c

Table 6.1: The input space for CRC.

Category	Input length (characters)	Execution flow
Small	0	CRC-1
	1	CRC-2
	2	CRC-3
Medium	10	CRC-4 & CRC-5
	46	CRC-6 & CRC-7
Large	99	CRC-8 & CRC-9

Table 6.2: The input space for SHA.

Category	Input length (characters)	Execution flow
Small	0	SHA-1
	1	SHA-2
	2	SHA-3
Medium	10	SHA-4 & SHA-5
	60	SHA-6 & SHA-7
Large	99	SHA-8 & SHA-9

of 1s and 0s. Since an integer is a 32-bit data type, the length of the input string is limited to 32 characters, see Table 6.4.

6.2 Software Implemented Fault Tolerance

In addition to the basic version of the workloads, we conducted experiments on the triple time redundant execution with forward recovery (TTR-FR) [129]. In TTR-FR, the target workload is executed three times and the result of each run is compared with the other two runs using a software implemented voter. If only one run of the program generates a different output, the output of the other two runs will be selected. Thus, the state of the faulty run moves forward to a fault-free point (forward recovery). If none of the outputs match, then error detection is signaled. The non-fault tolerance version of the workloads consists of three major code blocks;

Table 6.3: The input space for Qsort.

Category	Number of sorted elements	Execution flow
Sorted	6	Qsort-1
Mostly Sorted	4	Qsort-2 & Qsort-3
Partly Sorted	3	Qsort-4 & Qsort-5
	2	Qsort-6 & Qsort-7
Unsorted	0	Qsort-8 & Qsort-9

Table 6.4: The input space for BinInt.

Category	Input length (characters)	Execution flow
Small	0	BinInt-1
	9	BinInt-2 & BinInt-3
Medium	16	BinInt-4 & BinInt-5
	24	BinInt-6 & BinInt-7
Large	31	BinInt-8 & BinInt-9

startup, main function, and core function. In the TTR-FR implementation we add the voter to the main function to perform the majority voting. The core function, which is called three times from the main function, performs the foremost functionality of each workload. As an example, in Qsort, the sorting procedure is done in the Qsort's core function, whereas in CRC, the core function is responsible for the checksum calculations.

6.3 Experimental Setup and Fault Model

The workloads are executed on a Freescale MPC565 microcontroller, which implements a PowerPC architecture. Faults are injected into the microcontroller via a Nexus debug interface using Goofi-2 [17], a tool developed at Chalmers University. This environment allows us to inject faults, bit flips, into instruction set architecture (ISA) registers and main memory of the microcontroller. Ideally, the fault model to adopt for this evaluation should exhibit real faults, i.e., it should account for multiple and single bit

flips. However, there is no commonly agreed model for multiple bit flips. Thus, we adopt the single bit flip model as it has been done in other studies [114, 123, 125, 128, 131].

Faults are injected into the main memory (stack, data, etc.) and all CPU registers used by the execution flows. The registers include general purpose registers, program counter register, link register, integer exception register, and condition register. As the machine code of our workloads is stored in a Flash memory, it cannot be subjected to fault injection. We define fault in terms of time-location pair, where the location is a randomly selected bit in the memory word or CPU register, while the time corresponds to the execution of a given machine instruction (i.e., a point in the execution flow). A fault injection experiment consists of injecting one fault and observing its impact on a workload. A fault injection campaign is a series of fault injection experiments with a given execution flow.

6.4 Results

In this section, we present the outcomes of fault injection campaigns conducted on the 4 workloads. We carried out 9 campaigns per workload which resulted in a total of 36 campaigns for the basic version and 36 campaigns for the TTR-FR version. The campaigns consist of 25000 experiments except for CRC campaigns that are subjected to 12000 experiments. The error classification scheme of each experiment is:

- No Impact (NI), errors that do not affect the output of the execution flow.
- Detected by Hardware (DHW), errors that are detected by the hardware exceptions.
- Time Out (TO), errors that cause violation of the timeout.
- Value Failure (VF), erroneous output with no indication of failure (silent failure).
- Detected by Software (DSW), errors that are detected by the software detection mechanisms.
- Corrected by Software (CSW), errors that are corrected by the software correction mechanisms.

When presenting the results, we also refer to the coverage (COV) as the probability that a fault does not cause value failures, which is calculated in equation 6.1

$$COV = 1 - \#VF/N \tag{6.1}$$

Here N is the total number of experiments, and $\#VF$ is the total number of experiments that resulted in value failure. In addition to the experiments classified as detected by hardware, the coverage includes no impact and timeout experiments. No impact experiments can be the result of internal robustness of the workload; therefore they contribute to the overall coverage of the system. Experiments that are resulted in timeout are detected by Goofi-2. In a real application, watchdog timers are used to detect these types of errors.

6.4.1 Results for Workloads without Software Implemented Hardware Fault Tolerance

Tables 6.5, 6.6, 6.7, and 6.8 present failure distributions for all the workloads. Each row shows the percentage of experiments that fall in different error classifications. Due to the large number of experiments (25000 for SHA, BinInt, Qsort and 12000 for CRC), the 95% confidence interval for the measures in this section varies from ±0.08% to ±0.89%. For SHA and CRC, the percentage of experiments classified as value failures grows as the length of the inputs is increased.

If we consider that the value failure is distributed as a normal variable with a mean value equals to the quote between the number of value failure experiments and the total number of experiments, we can conduct one way analysis of variance (ANOVA). ANOVA is performed by testing the hypothesis H0 which states "there is no linear correlation between the length of the input and the percentage of value failure". The results of ANOVA in Table 6.9 allow us to reject H0 with a confidence of 95%. The reason behind this correlation is that when the length of the input increases, the number of reads from registers and memory locations are increased as well. Therefore, there are more possibilities to inject faults that result in value failure. Obviously, as the value failure increases linearly with the length, the coverage is linearly decreased (see Table 6.5 and Table 6.6).

Qsort and BinInt exhibit a non-linear variation of the value failure with,

Table 6.5: Failure distribution of all the execution flows of CRC (all values are in percentage).

Execution flow	NI	VF	DHW	TO	COV
CRC-1	42.7	6.1	48.2	3.0	93.9
CRC-2	32.9	17.9	46.7	2.4	82.1
CRC-3	28.3	24.3	45.8	1.6	75.7
CRC-4	20.8	34.3	44.0	0.8	65.7
CRC-5	20.3	35.5	43.6	0.6	64.5
CRC-6	17.1	39.6	43.0	0.3	60.4
CRC-7	16.6	39.8	43.4	0.2	60.2
CRC-8	15.7	41.2	42.7	0.4	58.8
CRC-9	16.0	41.9	41.8	0.3	58.1

Table 6.6: Failure distribution of all the execution flows of SHA (all values are in percentage).

Execution flow	NI	VF	DHW	TO	COV
SHA-1	18.9	38.8	41.0	1.4	61.2
SHA-2	17.8	40.1	41.0	1.1	59.9
SHA-3	17.6	40.8	40.6	1.0	59.2
SHA-4	16.8	42.1	39.7	1.4	57.9
SHA-5	15.9	43.1	39.4	1.6	56.9
SHA-6	11.5	47.1	39.5	1.9	52.9
SHA-7	11.4	47.7	39.3	1.6	52.3
SHA-8	10.7	48.8	38.8	1.7	51.2
SHA-9	10.7	49.1	38.4	1.8	50.9

respectively, the number of sorted elements and the input length (Table 6.7 and Table 6.8). For Qsort, this can be explained by considering that in addition to the number of sorted elements, the position of these elements impacts Qsort's behaviour. This causes different number of element comparisons and recursive calls to the core function. This effect is particularly evident for Qsort-4 and Qsort-5. Even though both have 50% of the input elements sorted, there is a difference of 4.85 percentage points between their

value failures. Although there is no linear correlation for Qsort, it is notable that the average value failures of the first five execution flows, which have more sorted elements, is 4.22 percentage points lower than the next four execution flows. BinInt, however, is a small program with an input space between 0 to 32 characters; these inputs for such a small application do not cause a significant variation in the failure distribution. Results in Table 6.9 show that the proportion of failures detected by the hardware exceptions is almost constant for a given workload (the coefficient is 0.019 for SHA, 0.04 for CRC, and 0.02 for BinInt). Analogously, the proportion of experiments classified as timeout is almost constant for all the workloads. It is worth noting that the startup code may vary in different systems. We therefore show the trend of value failures with/without the startup block in Fig. 6.1 and in Fig. 6.2. We can see that the trends in the two diagrams are similar which is due to the fact that the startup code consists of significantly fewer lines of code compared to the other blocks.

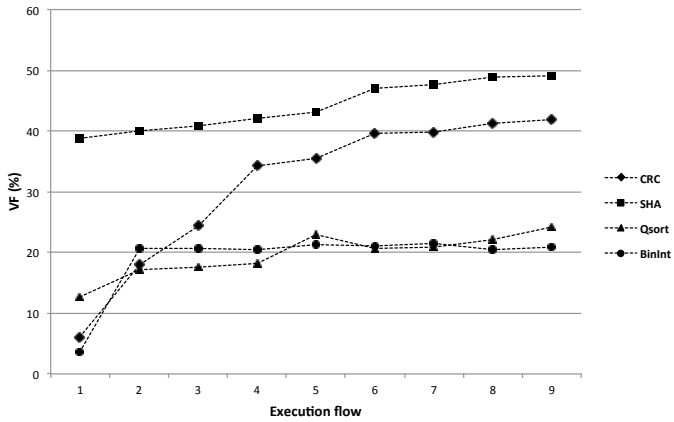

Figure 6.1: The percentage of value failures for different execution flows of each workload **with** the startup block.

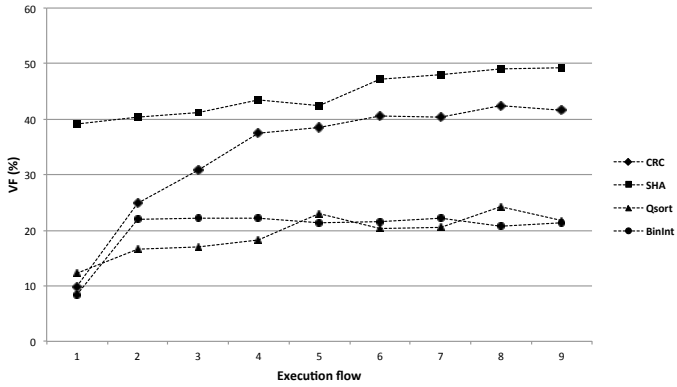

Figure 6.2: The percentage of value failures for different execution flows of each workload **without** the startup block.

Table 6.7: Failure distribution of all the execution flows of Qsort (all values are in percentage.

Execution flow	**NI**	**VF**	**DHW**	**TO**	**COV**
Qsort-1	37.1	12.7	46.8	3.5	87.3
Qsort-2	32.8	17.1	46.9	3.2	82.9
Qsort-2	32.8	17.1	46.9	3.2	82.9
Qsort-3	31.3	17.7	47.7	3.3	82.3
Qsort-4	31.7	18.1	46.8	3.9	81.9
Qsort-5	26.5	23.0	47.2	3.3	77.0
Qsort-6	29.0	20.7	46.0	4.3	79.3
Qsort-7	29.3	20.9	46.3	3.5	79.1
Qsort-8	27.2	22.1	46.6	4.2	77.9
Qsort-9	25.4	24.2	46.5	4.0	75.8

Table 6.8: Failure distribution of all the execution flows of BinInt (all values are in percentage).

Execution flow	NI	VF	DHW	TO	COV
BinInt-1	44.1	3.5	49.9	2.5	96.5
BinInt-2	34.9	20.6	41.5	3.0	79.4
BinInt-2	34.9	20.6	41.5	3.0	79.4
BinInt-3	34.7	20.6	41.6	3.1	79.4
BinInt-4	34.5	20.5	42.0	2.9	79.5
BinInt-5	35.3	21.2	40.5	3.0	78.8
BinInt-6	35.1	21.0	40.8	3.1	79.0
BitInt-7	34.8	21.5	40.5	3.2	78.5
BitInt-8	36.7	20.4	40.0	3.0	79.6
BinInt-9	35.5	20.9	40.5	3.1	79.1

Table 6.9: Null Hypothesis test results for the workloads.

Null Hypothesis(H0)	**Input Characteristic**	**Workload**	**p-value (α=0.05)**	**Result**	**Linear Regression Equation**
No linear correlation between VF and input characteristic	Length in characters	CRC	0.029	Reject	VF = 23.20 + 0.22length
		SHA	<0.001	Reject	VF = 40.64 + 0.09length
		BinInt	0.069	Accept	–
	Sorted elements	Qsort	0.053	Accept	–
No linear correlation between DHW and input characteristic	Length in characters	CRC	0.01	Reject	DHW = 45.84 - 0.04length
		SHA	0.034	Reject	DHW = 40.46-0.019length
		BinInt	0.02	Reject	DHW = 45.75 - 0.02length
	Sorted elements	Qsort	0.12	Accept	–
No linear correlation between TO and input characteristic	Length in characters	CRC	0.046	Reject	TO = 1.67 - 0.017length
		SHA	0.37	Accept	–
		BinInt	0.1	Accept	–
	Sorted elements	Qsort	0.18	Accept	–

6.4.2 Results for Workloads Equipped with TTR-FR

Table 6.10 presents the average results for the 9 execution flows of each workload. The percentage of value failures for SHA, CRC and BinInt is less than 2%, while for Qsort there is a higher percentage of value failures, about 5%. The proportion of value failure varies for different code blocks. With respect to the core function, the main contributor to the lack of coverage is faults in the program counter register. These faults change the control flow in such a way that the voter is incorrectly executed or not executed at all.

For instance, for the core function of SHA, around 96% of the value failures were caused by faults in the program counter register. Faults injected into the other code blocks, including the voter, are more likely to generate value failures since they are not protected by the TTR-FR. For Qsort, the relative size of the core function is smaller compared to the other programs. This resulted in only around 57% of the injections in this function, while in the other workloads more than 96% of faults were injected in the core function. This can explain the higher percentage of value failures in Qsort compared to the other workloads. In order to evaluate the robustness of the voter, we conducted exhaustive fault injections (i.e., we inject all possible faults) in the voter of each workload, see Table 6.11. It is notable that even though TTR-FR mechanism decreases the percentage of value failure, the voter is one of the main contributors to the occurrence of value failure. The average percentage of errors detected by the hardware exceptions does not vary significantly between the versions extended with TTR-FR and those without this mechanism for SHA, CRC, and BinInt, while it differs about 5% for Qsort.

Table 6.10: Average failure distributions for the workloads extended with TTR-FR, injections in all code blocks.

Workload	NI	VF	CSW	DSW	DHW	TO	COV
CRC	20.78	1.65	33.43	0.19	43.22	0.73	98.35
SHA	14.92	0.76	43.36	0.15	39.00	1.78	99.24
Qsort	28.74	5.42	20.37	0.77	41.89	2.79	94.58
BinInt	34.69	1.45	20.21	0.09	40.60	2.96	98.55

Table 6.11: Average failure distributions for the workloads extended with TTR-FR injection only in the voter code block.

Workload	VF
CRC	12.32
SHA	16.60
Qsort	17.05
BinInt	12.32

6.5 Input Selection

As we demonstrate in this work, the likelihood for a program to exhibit a value failure due to bit flips in CPU-registers or memory words depends on the input to the program. Thus, when we assess the error sensitivity of an executable program by fault injection, it is desirable to perform experiments with several inputs. In this section, we describe a method for selecting inputs such that they are likely to result in widely different outcome distributions. The selection process consists of three steps. First, the fault-free execution flows for a large set of inputs are profiled using assembly code metrics. We then use cluster analysis to form clusters of similar execution flows. Finally, we select one representative execution flow from each cluster and subject the workload to fault injection. We validate the method by showing that inputs in the same clusters indeed generate similar outcome distributions, while inputs in different clusters are likely to generate different outcome distributions.

6.5.1 Profiling

We adopt a set of 48 assembly metrics corresponding to different access types (read, write) to registers and memory sections along with various categories of assembly instructions (Table 6.12). Since that the 47 metrics might be redundant or highly correlated, with the *Principal Component Analysis* we select only a set of 6 uncorrelated metrics shown in Table 6.13.

For each group, we define the percentage of execution as the number of times that the instructions of that category are executed out of the total number of executed instructions. These 6 metrics are a proper representative

Table 6.12: The initial set of 48 assembly metrics.

Metric Number	Metric Name	Description
	General Metrics	
1	NEI	Number of different Executed Instructions, the total number of different instructions in the assembly code.
2	NE	Number of Executed instructions, i.e., the number of times that the PCR has been updated.
	Instruction Metrics	
3	NLI	Number of Load Instructions.
4	NSI	Number of Store Instructions.
5	NAI	Number of Arithmetic Instructions.
6	NBI	Number of Branch Instructions.
7	NLGI	Number of Logical Instructions.
8	NPI	Number of Processor Instructions.
9	PLI	Percentage of Load Instructions. (NLI/NE)
10	PSI	Percentage of Store Instructions. (NSI/NE)
11	PAI	Percentage of Arithmetic Instructions. (NAI/NE)
12	PBI	Percentage of Branch Instructions. (NBI/NE)
13	PLGI	Percentage of Logical Instructions. (NLGI/NE)
14	PPI	Percentage of Processor Instructions. (NPI/NE)
15	LAD	Load Distance, the average distance between two consecutive executions of load instructions.
16	SD	Store Distance, the average distance between two consecutive executions of store instructions.
17	AD	Arithmetic Distance, the average distance between two consecutive executions of arithmetic instructions.
18	BD	Branch Distance, the average distance between two consecutive executions of branch instructions.
19	LGD	Logical Distance, the average distance between two consecutive executions of logical instructions.
20	PD	Processor Distance, the average distance between two consecutive executions of processor instructions.
	Register Metrics	
21	NGPR	Total number of different GPRs accessed.
22	NRCR	Number of access in read mode to condition register.
23	NWCR	Number of access in write mode to condition register.
24	NRSP	Number of access in read mode to the stack pointer.
25	NWSP	Number of access in write mode to the Stack pointer.
26	NRGPR	Number of access in read mode to GPRs (all GPRs except r1, that has been counted in NRSP)
27	NWGPR	Number of access in write mode to GPRs? (all GPRs except r1, that has been counted in NWSP)
28	NRXER	Number of access in read mode to the XER.
29	RDCR	The average distance between two consecutive read operations from the CR.
30	WDCR	The average distance between two consecutive write operations? into the CR.
31	RDSP	The average distance between two consecutive read operations from the SP.
32	WDSP	The average distance between two consecutive write operations into the SP.
33	RDGPR	The average distance between two consecutive read operations from the GPRs.
34	WDGPR	The average distance between two consecutive write operations into the GPRs.
35	RDXER	The average distance between two consecutive read operations from the XER.
	Memory Metrics	
36	NRTXT	Number of times the program reads from the text section.
37	NRAS	Number of times the program reads from the Stack section.
38	NWAS	Number of times the program writes into the Stack section.
39	NRAB	Number of times the program reads from the bss/sbss section.
40	NWAB	Number of times the program writes into the bss/sbss section.
41	NRAD	Number of times that the program read from data/sdata section.
42	NWAD	Number of times the program writes into the data/sdata section.
43	RSD	The average distance between two consecutive read operations from the stack section in terms of PC executions.
44	WSD	The average distance between two consecutive write operations into the stack section in terms of PC executions.
45	RBD	The average distance between two consecutive read operations from the bss/sbss section in terms of PC executions.
46	WBD	The average distance between two consecutive write operations into the bss/sbss section in terms of PC executions.
47	RDD	The average distance between two consecutive read operations from the data/sdata section in terms of PC executions.
48	WDD	The average distance between two consecutive write operations into the data/sdata section in terms of PC executions.

of the metric set for our workloads. Therefore, these metrics are used as a *signature* for the fault-free run of each execution flow to be used in the clustering algorithm.

Table 6.13: Selected Assembly Metrics.

Categories	**Instructions**	**Metrics**
LOAD (LD)	lbz, li, lwi, lmw, lswi, ...	PLD (percentage of load instructions)
STORE (ST)	stb, stub, sth, sthx, stw,...	PST (percentage of store instructions)
ARITHMETIC (AI)	add, subf, divw, mulhw, ...	PAI (percentage of arithmetic instructions)
BRANCH (BR)	b, bl, bc, bclr, ...	PBR (percentage of branch instructions)
LOGICAL (LG)	and, or, cmp, rlwimi, ...	PLG (percentage of logical instructions)
PROCESSOR (PR)	mcrf, mftb, sc, rfi, ...	PPR (percentage of processor instructions)

6.5.2 Clustering

Cluster analysis divides the input set (the execution flow, in our case) into homogenous groups based on the measurements of input attributes, the signature of execution flows. We adopted the hierarchical clustering [132] due to the fact that unlike other clustering techniques (e.g., K-means), it does not require a preliminary knowledge of the number of clusters. Thus, we can validate a posteriori if the execution flows are clustered as expected. The hierarchical clustering adopted in this work evaluates the distance between two clusters according to the *centroid* method.

6.5.3 Input Selection Results

The clustering technique is applied to normalized values (mean equal to 0 and a variance equal to 1) of the assembly metrics. In the case of non-normalized

data, higher weights will be given to variables with higher variances. To prevent this effect, due to the significant variations in the metric values, e.g., the variance of PLD is orders of magnitude larger than the variance of PPR, we use the normalized values. Fig. 6.3 depicts dendrogram representations of the results of the clustering technique for the non-TTR-FR implementation of SHA, CRC, and Qsort workloads (BinInt has already shown a roughly constant variation in its failure distribution, thus, we exclude it from the clustering analysis). Each dendrogram is read from left to right.

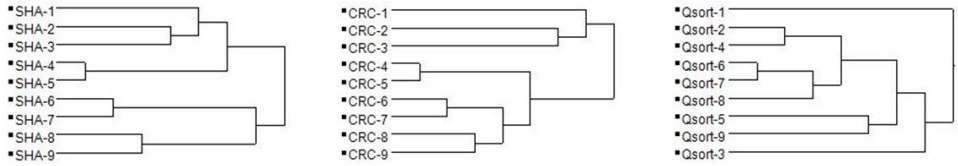

Figure 6.3: SHA, CRC and Qsort clusters on assembly metrics.

At the first stage of the algorithm, the execution flows of each workload are either grouped in 2-dimension clusters (e.g., SHA-4 and SHA-5) or left isolated (e.g., SHA-1). These groups can be easily linked to characteristics of the inputs in the case of SHA and CRC. Indeed, inputs with the same length (e.g., CRC-9 and CRC-8) or approximately the same length (e.g., CRC-2, CRC-3) belong to the same cluster. However in Qsort, this observation is not verified, since vectors with the same number of sorted elements are placed in different clusters (e.g., Qsort-8 and Qsort-9). At the next stage, different clusters are joined using vertical lines. The positions of these lines indicate the distance at which clusters are joined. In the case of our workloads, the algorithm groups the former clusters together by merging the inputs with "smaller size" (e.g., SHA-1, SHA-2, SHA-3 with SHA-4, SHA-5) and inputs with "larger size" (e.g., CRC-6, CRC-7 with CRC-8, CRC-9). In order to validate the results of our approach, we need to show that execution flows with a "similar" failure distribution belong to the same cluster. The same clustering algorithm can be used for identifying the execution flows that are similar in terms of failure distribution. This time, the error categories (VF, NI, DHW, TO) are used instead of the assembly metrics, see Fig. 6.4.

Comparing Fig. 6.3 and Fig. 6.4, for CRC and SHA, we can observe that the first clusters from the left are grouped exactly in the same way. For these workloads, after the profiling, we can arbitrarily select one execution

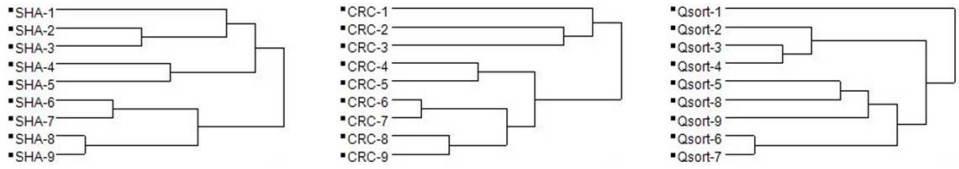

Figure 6.4: SHA, CRC and Qsort clusters on the failure distributions.

flow from each cluster for a fault injection campaign and consider its failure distribution as a representative of the other member of that cluster. In this way, the variation in failure mode distribution of a workload can be discovered by performing fault injection campaigns on fewer number of execution flows. We quantify the reduction, R, of fault injection campaigns in 6.2

$$R = (1 - C/I) * 100 \tag{6.2}$$

Here C indicates the number of clusters at the first stage, and I is the total number of execution flows. For CRC and SHA, the reduction is 45%, which means that we can save about 45% of time. Hence, for these workloads we can profile their execution flows and on the basis of the obtained clusters decide whether to conduct a fault injection campaign or not. It is notable that input selection requires very limited human interactions and it is mostly accomplished by a fault-free run of the execution flow performed by Goofi-2, a signature extractor tool, and a data analysis tool. In our experimental environment, profiling costs up to 5 hours, while a fault injection campaign costs up to 2 days. This is a significant benefit of the proposed approach. For Qsort there is no mapping between the clusters in the assembly space and the ones for the failure distribution. This might mean that for some applications like Qsort, where the failure distribution is dependent on more than just the length of input, other suitable assembly metrics are required. We exclude that this result is tied to the choice of the clustering method since we also obtain identical results with other methods such as *average* and *ward*.

Chapter 7

Conclusions and Future Work

Robustness evaluation is a fundamental activity for software systems adopted in mission critical systems. Fault injection is an attractive means to assess the robustness, however, its application is still costly and cumbersome. In this thesis, we focused on two fault injection techniques, namely Software Implemented Fault Injection (SWIFI) and Robustness Testing (RT). For both techniques we show that the workload plays a crucial role since it can affect the experiment outcome at large and increase the cost due to their application.

In complex systems such as an operating system which have different states, the workload can induce transition from one state to another. RT has been applied to an operating system used in the avionics domain for which we investigated the impact of OS state through experiments on the File System. With the *Approach I*, we included the OS state in the robustness test plan through a model of the File System, which encompasses a set of factors (such as file tree layout and concurrent I/O operations) that are most influential on the File System behavior, and that can be controlled by the tester. We performed an experiment using the proposed model, which highlighted the influence of the OS state on the test outcomes and on statement coverage.

In particular, robustness tests were able to reach corner cases with complex interactions with other subsystems (such as scheduling, caching and memory management), which are not covered by traditional robustness testing. In turn, this approach comes in handy to achieve an increased confidence in OS robustness with low human effort, since both robustness test cases and OS states can be automatically generated once programmed by the tester.

Alternatively, we conceived *Approach II*, also named *SABRINE*, which infers behavioral models of the OS in an automatic fashion and does not require the tester to know internal details of the OSs. The behavioral models are then used for creating and executing a robustness test suite. We compared the results obtained with SABRINE against random testing, which is often considered as a comparison baseline. Results clearly showed that SABRINE outperforms random strategy, as the number of test cases can be dramatically reduced while detecting the same robustness vulnerabilities.

Table 7.1 compares qualitatively the two approaches in terms of fault injection properties that we introduced in Section 2.3.1. Both approaches reach high reproducibility, since that we have not measured the reproducibility of the Approach I (as we did for Approach II), we are conservative and assign a medium level. Approaches present a high controllability because both allows to select when and where to inject an error. The temporal intrusiveness for Approach I is low compared to Approach II which requires to log the interactions of the component under test. Spatial intrusiveness, in principle is none for Approach I because no additional code is necessary for the execution of the tests. Differently, Approach II instruments the operating system in order to log the interactions.

Test effort indicates the work required to the tester during the "setup" of the approach (e.g., for developing the model of the target or selecting the call points). Aside from the setup, both approaches executes automatically therefore no additional effort is needed. The test effort of the Approach I can vary from medium to high because the model of the component under test is manually developed and he/she needs knowledge on the characterizes of the target (e.g., a file system can be balanced or unbalanced, tuning of the model attributes). Likewise, this activity should be conducted by experienced testers to avoid erroneous model of the target. Conversely, Approach II automatically extracts the behavioral models. During the setup the tester has to only indicate the call points which can be automatically extract with simple scripts.

In this work SABRINE creates behavioral models under the stimulation of a specific workload. In the future, more workloads should be used and select by avoiding workloads that generate the same behavioral models or it might be possible to use a state setter as shown in Approach I.

Regarding SWIFI, we preliminary investigated how the input affects the failure distribution, an aspect often neglected in the fault injection planning.

Table 7.1: Comparison of Approach I and Approach II (SABRINE).

Property	**Approach I**	**Approach II (SABRINE)**
Repeatability	Medium	High
Controllability	High	High
Temporal Intrusiveness	Low	Low to Medium
Spatial Intrusiveness	None	Low to Medium
Tester Effort	Medium to High	Low

The experiments, carried out on an embedded system, demonstrate that for some applications, the size of input is linearly correlated to the percentage of value failure while the percentage of faults detected by the hardware exceptions is workload dependent, i.e., it is not affected by the input. As similar inputs (e.g., same length inputs) result in a similar failure distribution, we devised an approach to reduce the number of fault injections. In addition, we studied assembly level metrics with respect to the failure distribution. While in performance benchmarking some study [133] explores the correlation between metrics and performance factors (e.g., power consumption), in the dependability field there is a no investigation on this area. Future researches might focus on prediction algorithms that allow to estimate the failure distribution of a specific workload as the input varies.

Bibliography

[1] United States General Accounting Office. Stronger Management Practices Are Needed to Improve DOD's Software-Intensive Weapon Acquisitions. Technical report, Department of Defence, 2004.

[2] D. Michaels and A. Pasztor. Incidents Prompt New Scrutiny Of Airplane Software Glitches. 2006.

[3] M. Torchiano and M. Morisio. Overlooked aspects of COTS-based development. *Software, IEEE*, 21(2):88 – 93, march-april 2004.

[4] C.P. Shelton, P. Koopman, and K. Devale. Robustness testing of the Microsoft Win32 API. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 261 –270, 2000.

[5] RTCA, EUROCAE. Software Considerations in Airbone Systems and Equipment Certification DO178B, 1992.

[6] RTCA, EUROCAE. Software Considerations in Airbone Systems and Equipment Certification DO178C, 2011.

[7] CENELEC. EN50126, 1999.

[8] International Organization for Standardization. Product development: software level, 2006.

[9] E. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly good software can behave. *Software, IEEE*, 14(4):73 –83, jul/aug 1997.

[10] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10 – 16, nov.-dec. 2005.

[11] R. Baumann. Soft errors in advanced computer systems. *Design Test of Computers, IEEE*, 22(3):258 – 266, may-june 2005.

[12] Lions, J. Ariane 5 Flight 501 Failure. Technical report, European Space Agency, 2007.

[13] S.S. Mukherjee, J. Emer, and S.K. Reinhardt. The soft error problem: an architectural perspective. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 243 – 247, feb. 2005.

[14] 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. *Software Engineering, IEEE Transactions on*, 16(2):166 –182, feb 1990.

[15] U. Gunneflo, J. Karlsson, and J. 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, jun 1989.

[16] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. FERRARI: a flexible software-based fault and error injection system. *Computers, IEEE Transactions on*, 44(2):248 –260, feb 1995.

[17] D. Skarin, R. Barbosa, and J. Karlsson. Goofi-2: A tool for experimental dependability assessment. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 557 –562, 28 2010-july 1 2010.

[18] D.T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R.K. Iyer. Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International*, pages 91 –100, 2000.

[19] Jiantao Pan, P. Koopman, Yennun Huang, R. Gruber, and Mimi Ling Jiang. Robustness testing and hardening of corba orb implementations. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 141 –150, july 2001.

[20] Wei-Lun Kao and R.K. Iyer. Define: a distributed fault injection and monitoring environment. In *Fault-Tolerant Parallel and Distributed Systems, 1994., Proceedings of IEEE Workshop on*, pages 252 –259, jun 1994.

[21] N.P. Kropp, P.J. Koopman, and D.P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 230 –239, jun 1998.

[22] Manuel Rodriguez, Jean-Charles Salles, Fabre, and Jean Arlat. MAFALDA: Microkernel Assessment by Fault Injection and Design Aid. In Jan Hlavicka, Erik Maehle, and Andras Pataricza, editors, *Dependable Computing EDCC-3*, volume 1667 of *Lecture Notes in Computer Science*, pages 143–160. Springer Berlin Heidelberg, 1999.

[23] Marco Vieira and Henrique Madeira. A dependability benchmark for OLTP application environments. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 742–753. VLDB Endowment, 2003.

[24] J.-C. Ruiz, P. Yuste, P. Gil, and L. Lemus. On benchmarking the dependability of automotive engine control applications. In *Dependable Systems and Networks, 2004 International Conference on*, pages 857–866, 2004.

[25] M. Pezze and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley,John & Sons, 2007.

[26] P. Koopman and J. DeVale. The exception handling effectiveness of POSIX operating systems. *Software Engineering, IEEE Transactions on*, 26(9):837–848, 2000.

[27] A. Johansson, N. Suri, and B. Murphy. On the selection of error model(s) for OS robustness evaluation. In *Proc. Intl. Conf. on Dependable Systems and Networks*, pages 502–511, 2007.

[28] M. Rebaudengo, M. Sonxa Reorda, and M. Viaolante. A new approach to software-implemented fault tolerance. In *IEEE Latin American Test Workshop*, 2004.

[29] R. Alexandersson and J. Karlsson. Fault injection-based assessment of aspect-oriented implementation of fault tolerance. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 303 –314, june 2011.

[30] A. Johansson, N. Suri, and B. Murphy. On the impact of injection triggers for OS robustness evaluation. In *Proc. Intl. Symp. on Software Reliability Engineering*, pages 127–126, 2007.

[31] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. Fault injection experiments using FIAT. *Computers, IEEE Transactions on*, 39(4):575 –582, apr 1990.

[32] C. Sarbu, A. Johansson, N. Suri, and N. Nagappan. Profiling the operational behavior of os device drivers. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 127 –136, nov. 2008.

[33] U.S. Department of Transportation Federal Aviation Administration, Office of Aviation Research Washington, D.C. 20591. Commercial Off-The-Shelf (COTS) Avionics Software Study, DOT/FAA/AR-01/26. Technical report, 2001.

[34] IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology, 1990.

[35] H. Madeira, R.R. Some, F. Moreira, D. Costa, and D. Rennels. Experimental evaluation of a COTS system for space applications. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 325 – 330, 2002.

[36] VV. AA. *State of the art of AMBER project*. 2009.

[37] J. Durães and H. Madeira. Characterization of Operating Systems Behavior in the Presence of Faulty Drivers through Software Fault Emulation. In *Proc. Pacific Rim Intl. Symp. on Dependable Computing*, pages 201–209, 2002.

[38] Natella, R. and Cotroneo, D. and Duraes, J.A. and Madeira, H.S. On Fault Representativeness of Software Fault Injection. *Software Engineering, IEEE Transactions on*, 39(1):80 –96, jan. 2013.

[39] R. Natella. *Achieving Representative Faultloads in Software Fault Injection*. PhD thesis, Università degli Studi di Napoli Federico II, 2011.

[40] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proc. ACM Symp. on Operating Systems Principles*, 2001.

[41] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75 –82, apr 1997.

[42] J. Gray. Why do computers stop and what can be done about it? In *Proc. of the Symposium on Reliability in Distributed Software and Database Systems (SRDS)*, pages 3–12, 1986.

[43] A. Shahrokni and F. Robert. A systematic review of software robustness. *Information and Software Technology*, 55(1):1 – 17, 2013.

[44] B.P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[45] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical Report CSTR-95-1268, 1998.

[46] A.K. Ghosh, M. Schmid, and V. Shah. Testing the Robustness of Windows NT Software. In *Proc. Intl. Symp. on Software Reliability Engineering*, pages 231–235, 1998.

[47] C.P. Dingman, J. Marshall, and D.P. Siewiorek. Measuring robustness of a fault tolerant aerospace system. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 522–527. IEEE, 1995.

[48] P. Koopman and J. DeVale. Comparing the Robustness of POSIX Operating Systems. In *Proc. Intl. Symp. on Fault-Tolerant Comp.*, pages 30–37, 1999.

[49] IEEE. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX) Part 1. *IEEE Std 1003.1b-1993*, 1994.

[50] K. Kanoun, Y. Crouzet, A. Kalakech, A.E. Rugina, and P. Rumeau. Benchmarking the Dependability of Windows and Linux Using PostMark Workloads. In *Proc. Intl. Symp. on Fault-Tolerant Comp.*, pages 11–20, 2005.

[51] A. Kalakech, K. Kanoun, Y. Crouzet, and J. Arlat. Benchmarking the Dependability of Windows NT4, 2000 and XP. In *Proc. Intl. Conf. on Dependable Systems and Networks*, pages 681–686, 2004.

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

[53] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *Proc. USENIX Large Installation System Administration Conf.*, pages 101–111, 2006.

[54] A. Albinet, J. Arlat, and J.C. Fabre. Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel. In *Proc. Intl. Conf. on Dependable Systems and Networks*, pages 867–876, 2004.

[55] J. Durães, M. Vieira, and H. Madeira. Multidimensional Characterization of the Impact of Faulty Drivers on the Operating Systems Behavior. *IEICE Trans. on Information and Systems*, 86(12):2563–2570, 2003.

[56] A. Johansson, N. Suri, and B. Murphy. On the selection of error model(s) for OS robustness evaluation. In *Proc. Intl. Conf. on Dependable Systems and Networks*, pages 502–511, 2007.

[57] S. Winter, C. Sârbu, N. Suri, and B. Murphy. The impact of fault models on software robustness evaluations. In *Proc. Intl. Conf. on Software Engineering*, pages 51–60, 2011.

[58] W. Gu, Z. Kalbarczyk, R.K. Iyer, Z. Yang, et al. Characterization of linux kernel behavior under errors. In *Proc. Intl. Conf. on Dependable Systems and Networks*, pages 459–468, 2003.

[59] L.N. Bairavasundaram, M. Rungta, N. Agrawa, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, and M.M. Swift. Analyzing the effects of disk-pointer corruption. In *Proc. IEEE Intl. Conf. Dependable Systems and Networks*, pages 502–511. IEEE, 2008.

[60] A. Johansson, N. Suri, and B. Murphy. On the impact of injection triggers for OS robustness evaluation. In *Proc. Intl. Symp. on Software Reliability Engineering*, pages 127–126, 2007.

[61] C. Sârbu, A. Johansson, N. Suri, and N. Nagappan. Profiling the operational behavior of OS device drivers. *Empirical Software Engineering*, 15(4):380–422, 2010.

[62] V. Prabhakaran, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. Model-based failure analysis of journaling file systems. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 802–811. IEEE, 2005.

[63] Perry Groot, Frank Harmelen, and AnnetteTen Teije. Torture Tests: A Quantitative Analysis for the Robustness of Knowledge-Based Systems. In Rose Dieng and Olivier Corby, editors, *Knowledge Engineering and Knowledge Management Methods, Models, and Tools*, volume 1937 of *Lecture Notes in Computer Science*, pages 403–418. Springer Berlin Heidelberg, 2000.

[64] ZoltÃ¡n Micskei, IstvÃ¡n Majzik, and Francis Tam. Comparing Robustness of AIS-Based Middleware Implementations. In Miroslaw Malek, Manfred Reitenspie, and Aad Moorsel, editors, *Service Availability*, volume 4526 of *Lecture Notes in Computer Science*, pages 20–30. Springer Berlin Heidelberg, 2007.

[65] A. Kovi and Z. Micskei. Robustness testing of standard specifications-based ha middleware. In *Distributed Computing Systems Workshops (ICDCSW), 2010 IEEE 30th International Conference on*, pages 302–306, 2010.

[66] Antonio Bovenzi, Aniello Napolitano, Christian Esposito, and Gabriella Carrozza. Jfit: an automatic tool for assessing robustness of dds-compliant middleware. In Domenico Cotroneo, editor, *Innovative Technologies for Dependable OTS-Based Critical Systems*, pages 69–81. Springer Milan, 2013.

[67] Tsanchi Li, C.-M. Chen, B. Horgan, Ming Yee Lai, and S.Y. Wang. A software fault insertion testing methodology for improving the robustness of telecommunications systems. In *Communications, 1994. ICC '94, SUPERCOMM/ICC '94, Conference Record, 'Serving Humanity Through Communications.' IEEE International Conference on*, pages 1767–1771 vol.3, 1994.

[68] Ilaria Canova Calori, Tor Stålhane, and Sven Ziemer. Robustness analysis using fmea and bbn case study for a web-based application. 2007.

[69] Chen Fu, Barbara Ryder, Ana Milanova, and David Wonnacott. Testing of java web services for robustness. In *In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA*, pages 23–34. ACM Press, 2004.

[70] Samer Hanna and Malcolm Munro. An approach for wsdl-based automated robustness testing of web services. In Chris Barry, Michael Lang, Wita Wojtkowski, Kieran Conboy, and Gregory Wojtkowski, editors, *Information Systems Development*, pages 1093–1104. Springer US, 2009.

[71] S. Hanna and M. Munro. Fault-based web services testing. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, pages 471–476, 2008.

[72] N. Laranjeiro, R. Oliveira, and M. Vieira. Applying text classification algorithms in web services robustness testing. In *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, pages 255–264, 31 2010-Nov. 3.

[73] M. Susskraut and C. Fetzer. Robustness and security hardening of cots software libraries. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 61–71, 2007.

[74] K.Z. Zamli, M.D.A. Hassan, N.A.M. Isa, and S.N. Azizan. An automated software fault injection tool for robustness assessment of java cots. In *Computing Informatics, 2006. ICOCI '06. International Conference on*, pages 1–6, 2006.

[75] A. Tarhini, A. Rollet, and H. Fouchal. A pragmatic approach for testing robustness on real-time component based systems. In *Computer Systems and Applications, 2005. The 3rd ACS/IEEE International Conference on*, pages 143–, 2005.

[76] A. Vasan and A.M. Memon. Aspire: automated systematic protocol implementation robustness evaluation. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 241–250, 2004.

[77] Chuanming Jing, Zhiliang Wang, Xia Yin, and Jianping Wu. A formal approach to robustness testing of network protocol. In Jian Cao, Minglu Li, Min-You Wu, and Jinjun Chen, editors, *Network and Parallel Computing*, volume 5245 of *Lecture Notes in Computer Science*, pages 24–37. Springer Berlin Heidelberg, 2008.

[78] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. Software aging analysis of the linux operating system. *2010 IEEE 21st International Symposium on Software Reliability Engineering*, 0:71–80, 2010.

[79] A. Bovenzi, D. Cotroneo, R. Pietrantuono, and S. Russo. Workload characterization for software aging analysis. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 240–249, 2011.

[80] R. McDougall, J. Mauro, and B. Gregg. *Solaris™ Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Prentice Hall PTR, 2006.

[81] B. Jacob, P. Larson, B. Leitao, and S.A.M.M. da Silva. SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems. *IBM Redbook*, 2008.

[82] M.E. Russinovich and A. Margosis. *Windows® Sysinternals Administrator's Reference*. Microsoft Press, 2011.

[83] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1), 2012.

[84] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis*. 11th edition, 2006.

[85] R. Kannan, S. Vempala, and A. Veta. On clusterings-good, bad and spectral. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 367–, Washington, DC, USA, 2000. IEEE Computer Society.

[86] Ulrike von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, 2007.

[87] F. R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.

[88] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1), January 2002.

[89] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, 2009.

[90] Lothar Wendehals and Alessandro Orso. Recognizing behavioral patterns at runtime using finite automata. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, WODA '06, 2006.

[91] L. Mariani and M. Pezzè. Dynamic detection of COTS component incompatibility. *Software, IEEE*, 24(5):76–85, 2007.

[92] L. Mariani, F. Pastore, and M. Pezzè. Dynamic analysis for diagnosing integration faults. *Software Engineering, IEEE Transactions on*, 37(4):486–508, 2011.

[93] S. Garg, A. van Moorsel, K. Vaidyanathan, and K.S. Trivedi. A methodology for detection and estimation of software aging. In *Proc. Intl. Symposium on Software Reliability Engineering*, pages 283–292. IEEE, 1998.

[94] Akinobu Mita. Fault injection capabilities infrastructure. `http://www.kernel.org/doc/Documentation/fault-injection/fault-injection.txt`.

[95] D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[96] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389 – 398, 2002.

[97] M. Favalli and C. Metra. Optimization of error detecting codes for the detection of crosstalk originated errors. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pages 290 –296, 2001.

[98] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Case for Lifetime Reliability-Aware Microprocessors. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 276–. IEEE Computer Society, 2004.

[99] R.J. Martinez, P.J. Gil, G. Martin, C. Perez, and J.J. Serrano. Experimental validation of high-speed fault-tolerant systems using physical fault injection. In *Dependable Computing for Critical Applications 7, 1999*, pages 249 –265, nov 1999.

[100] H. Madeira, M. Rela, F. Moreira, and J. G. Silva. RIFLE: A general purpose pin-level fault injector. In *Dependable Computing EDCC-1*, volume 852 of *Lecture Notes in Computer Science*, pages 197–216. 1994.

[101] J. Arlat, Y. Crouzet, and J.-C. Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*, pages 348 –355, jun 1989.

[102] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI: generic object-oriented fault injection tool. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 83 –88, july 2001.

[103] P. Yuste, D. de Andres, L. Lemus, J.J. Serrano, and P. Gil. INERTE: integrated nexus-based real-time fault injection tool for embedded systems. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, page 669, june 2003.

[104] Alfredo Benso, Maurizio Rebaudengo, and MatteoSonza Reorda. FlexFi: A Flexible Fault Injection Environment for Microprocessor-Based Systems. In Massimo Felici and Karama Kanoun, editors, *Computer Safety, Reliability and Security*, volume 1698 of *Lecture Notes in Computer Science*, pages 323–335. Springer Berlin Heidelberg, 1999.

[105] G. Miremadi and J. Torin. Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection. *Reliability, IEEE Transactions on*, 44(3):441 –454, 1995.

[106] J. Karlsson, U. Gunneflo, P. Liden, and J. Torin. Two fault injection techniques for test of fault handling mechanism. In *Test Conference, 1991, Proceedings., International*, page 140, oct 1991.

[107] A. Rajabzadeh, S. Ghassem, and M. Miremadi. Experimental Evaluation of Master/Checker Architecture Using Power Supply- and Software-Based Fault Injection. *11th IEEE International On-Line Testing Symposium*, 0:239, 2004.

[108] D. Skarin. *On Fault Injection-Based Assessment of Safety Critical Systems*. PhD thesis, Chalmers University of Technology, 2010.

[109] D. Skarin, R. Barbosa, and J. Karlsson. Comparing and Validating Measurements of Dependability Attributes. In *Dependable Computing Conference (EDCC), 2010 European*, pages 3 –12, april 2010.

[110] P. Troger, F. Salfner, and S. Tschirpke. Software-Implemented Fault Injection at Firmware Level. In *Dependability (DEPEND), 2010 Third International Conference on*, pages 13 –16, july 2010.

[111] W.-I. Kao, R.K. Iyer, and D. Tang. FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults. *Software Engineering, IEEE Transactions on*, 19(11):1105 –1118, nov 1993.

[112] SeungJae Han, K.G. Shin, and H.A. Rosenberg. DOCTOR: an integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 204 –213, apr 1995.

[113] T.K. Tsai, R.K. Iyer, and D. Jewitt. An approach towards benchmarking of fault-tolerant commercial systems. In *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, pages 314 –323, jun 1996.

[114] J. Carreira, H. Madeira, and J.G. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *Software Engineering, IEEE Transactions on*, 24(2):125 –136, feb 1998.

[115] A. Benso, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. EXFI: a low-cost fault injection system for embedded microprocessor-based boards. *ACM Trans. Des. Autom. Electron. Syst.*, 3(4):626–634, October 1998.

[116] Antonio Dasilva, José-F Martínez, Lourdes López, Ana-B García, and Luis Redondo. Exhaustif: a fault injection tool for distributed heterogeneous embedded systems. In *Proceedings of the 2007 Euro American conference on Telematics and information systems*, EATIS '07, pages 17:1–17:8, New York, NY, USA, 2007. ACM.

[117] T.K. Tsai, R.K. Iyer, and D. Jewitt. An approach towards benchmarking of fault-tolerant commercial systems. In *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, pages 314 –323, jun 1996.

[118] D. Audet, S. Masson, and Y. Savaria. Reducing fault sensitivity of microprocessor-based systems by modifying workload structure. In *Defect and Fault Tolerance in VLSI Systems, 1998. Proceedings., 1998 IEEE International Symposium on*, pages 241–249, 1998.

[119] Peter Folkesson and Johan Karlsson. Considering workload input variations in error coverage estimation. In *Proceedings of the Third European Dependable Computing Conference on Dependable Computing*, EDCC-3, pages 171–190, London, UK, UK, 1999. Springer-Verlag.

[120] N. Oh, M. Subahshish, and McCluskey E. J. ED^4: error detection by diverse data and duplicated instructions. *IEEE Transaction on Computer*, 51:626–634, 2002.

[121] H. Madeira and J.G. Silva. On-line signature learning and checking: experimental evaluation. In *CompEuro '91. Advanced Computer Technology, Reliable Systems and Applications. 5th Annual European Computer Conference. Proceedings.*, pages 642–646, 1991.

[122] M. Rebaudengo, M. Sonza Reorda, and M.; Violante. New Approach to Software-Implemented Fault Tolerance. In Jan Hlavicka, Erik Maehle, and Andras Pataricza, editors, *Journal of Electronic Testing*, volume 20 of *Lecture Notes in Computer Science*, pages 433–437. Springer Berlin Heidelberg, 2004.

[123] M. Rebaudengo, M.S. Reorda, and M. Violante. A new software-based technique for low-cost fault-tolerant application. In *Reliability and Maintainability Symposium, 2003. Annual*, pages 25–28.

[124] B. Nicolescu, Y. Savaria, and R. Velazco. Software detection mechanisms providing full coverage against single bit-flip faults. *Nuclear Science, IEEE Transactions on*, 51(6):3510–3518, 2004.

[125] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. SWIFT: software implemented fault tolerance. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 243–254, March.

[126] J. Vinter, A. Johansson, P. Folkesson, and J. Karlsson. On the design of robust integrators for fail-bounded control systems. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pages 415–424, June.

[127] D. Skarin and J. Karlsson. Software Implemented Detection and Recovery of Soft Errors in a Brake-by-Wire System. In *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*, pages 145–154, May.

[128] J.J. Cook and C. Zilles. A characterization of instruction-level error derating and its implications for error detection. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 482–491, 2008.

[129] R. Alexandersson and J. Karlsson. Fault injection-based assessment of aspect-oriented implementation of fault tolerance. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 303–314, 2011.

[130] Raul Barbosa, Jonny Vinter, Peter Folkesson, and Johan Karlsson. Assembly-level preinjection analysis for improving fault injection efficiency. In *in Proceedings of the Fifth European Dependable Computing Conference (EDCC-5*, 2005.

[131] A. Martinez-Alvarez, S. Cuenca-Asensi, F. Restrepo-Calle, F. Palomo Pinto, H. Guzman-Miranda, and M. A. Aguirre. Compiler-Directed Soft Error Mitigation for Embedded Systems. *IEEE Transactions on Dependable and Secure Computing*, 9(2):159–172, 2012.

[132] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, September 1999.

[133] L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pages 2–12, 2005.