1 2 9 0

UNIVERSIDADE D COIMBRA

Bernardo Marques Graça

Accelerating fault injection campaigns using failure models

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor Frederico Cerveira, co-advised by Professor Henrique Madeira, and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July of 2023

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Bernardo Marques Graça

Accelerating fault injection campaigns using failure models

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Frederico Cerveira, co-advised by Prof. Henrique Madeira and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July of 2023

Abstract

Computer systems are becoming increasingly complex and being used for various tasks, some of which are critical. For a trustable service delivery, the system must be able to avoid or tolerate failures that may occur during its execution. Fault injection is an experimental technique for the validation of a system and its fault-handling mechanisms.

Campaigns involving Fault Injection (FI) may last months, and care must be taken during their planning in order not to produce unreliable results. Thus, improving fault injection efficiency through approaches that aim to accelerate the experiments without losing representativeness is important.

The goal of this dissertation is the acceleration of fault injection campaigns. For that purpose, we developed a technique for injecting failure models. Failure models are the outcome of a fault like a crash or hang that can affect a system. To validate the technique, we performed experiments in a virtualized setup in order to compare our injection technique with traditional FI.

We started by studying the state of the art of dependability and FI to understand the approaches used for FI acceleration and support our choices and help us define our technique. We performed 1739 experiments over three months, and the results obtained from our validation show that the injection of crash failures in the hypervisor can replace hardware FI when we aim to analyze specific metrics, producing failure results almost 3x faster. The hang failure injection is also a valid alternative to hardware FI when we want to study the manifestation latency. The results show that injecting failures can be an effective approach to evaluate the performance of fault tolerance mechanisms, however it is not a good alternative for evaluating the dependability of a system or for designing failure prediction mechanisms

Keywords

Dependability, Fault injection, Failure injection, Fault injection tools, Fault injection acceleration, Hardware faults, Software faults, Virtualization.

Resumo

Os sistemas informáticos estão a tornar-se cada vez mais complexos e a ser usados para variadas tarefas, sendo algumas delas consideradas críticas. Para um fornecimento fiável do serviço, o sistema deve conseguir evitar ou tolerar avarias que possam ocorrer durante a sua execução. Injeção de falhas é uma técnica baseada em experiências, usada para a validação de um sistema e dos seus mecanismos de tratamento de falhas.

As campanhas de injeção de falhas podem durar vários meses e deve-se ter cuidado durante o seu planeamento de modo a que não produza resultados não fiaveis. Assim, é importante a melhoria da eficiência de injeção de falhas através de abordagens que tenham o objetivo de acelerar as experiências sem a perda de representatividade.

O objetivo desta dissertação é a aceleração de campanhas de injeção de falhas. Para esse propósito, desenvolvemos uma técnica para a injeção de modelos de avarias de maneira a acelerar a validação de um sistema. Um modelo de avaria é o resultado de uma falha como um crash ou hang que pode afetar o sistema. Para validar a técnica, efetuamos experiências num sistema virtualizado para comparar a nossa abordagem com a injeção de falhas tradicional.

Começamos por realizar um estudo acerca do estado da arte da confiabilidade e da injeção de falhas para se perceber algumas abordagens e também para apoiar as escolhas efetuadas na implementação da nossa técnica. Realizamos 1739 experiências ao longo de três meses e os resultados obtidos a partir da nossa validação mostrou que a injeção de crashes no hypervisor pode substituir a injeção de falhas de hardware quando temos como objetivo estudar certas métricas, produzindo resultados de avarias aproximadamente 3x mais rapido. A injeção de hangs pode também ser uma alternativa à injeção de falhas de hardware quando queremos estudar a latência da manifestação da avaria. Assim a injeção de avarias pode ser uma abordagem eficiente quando o objetivo é avaliar um mecanismo de tolerância de falhas no entanto não é uma boa alternativa quando se quer avaliar a confiabilidade de um sistema ou para desenhar mecanismos de previsão de avarias.

Palavras-Chave

Confiabilidade, Injeção de falhas, Injeção de avarias, Ferramentas de injeção de falhas, Aceleração de Injeção de Falhas, Falhas de hardware, Falhas de software, Virtualização

Acknowledgements

First, I would like to thank my parents for all the efforts and sacrifices they made during this academic stage of mine and who supported me through all the good and bad times. I want to thank my sister for her support. I also thank my girlfriend for all her support and motivation throughout this work and for believing in me.

I also want to thank my advisor Frederico Cerveira because, without him, this work would not have been possible. I appreciate all the attention and help he gave me at all times and all the teachings, having always been ready to answer and solve my doubts. I also thank my co-advisor Henrique Madeira for his help and suggestions during the development of this work.

I thank all my family for their support through these five years and for the motivation they gave me. Finally, I want to thank my friends for all the good times and fun we spent together, for all the help in this phase, and for the moments of work we spent together.

Contents

Acronyms

DTI	Data Type Identification
DUE	Detected Unrecoverable Error
FI	Fault Injection
HFI	Hardware Fault Injection
HWIFI	Hardware Implemented Fault Injection
ISA	Instruction Set Architecture
ODC	Orthogonal Defect Classification
SDC	Silent Data Corruption
SEU	Single Event Upset
SFI	Software Fault Injection
SWIFI	Software-Implemented Fault Injection
VM	Virtual Machine
VMs	Virtual Machines

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, we are becoming more dependent on hardware-heavy and software-heavy systems for various services. The constant technological innovation and the application of new technologies in multiple tasks, from the most simple to the most critical, means that systems are becoming more complex, involving multiple components and integrating different features. Therefore, ensuring that a system does not fail and consequently stops providing the correct service is essential.

FI is an important technique for evaluating dependability that can generate realistic results faster than through the natural occurrence of faults during the service deliverance of a system. In this technique, faults are injected into a system in order to study its behavior and understand how we can improve its reliability. FI is a vast research field that can be useful in different scenarios, and there are already multiple techniques targeting different systems and goals. However, despite the existing research, there are still limitations regarding the acceleration of FI.

1.1 Motivation

FI can be seen as a viable solution for evaluating systems and their components however, it is an experimental process that can be slow and last several months in order to perform an evaluation. This drawback is a consequence of the complexity of FI may present since there are multiple variables to be considered, such as the fault type, where it should be injected, and the time of injection [1].

Therefore it is essential to improve the efficiency of FI through new research and approaches, with the aim of keeping the consistency and representativeness of results from the experiments. There are already multiple techniques and research on this topic. However, there are still problems associated with the acceleration of FI.

This dissertation was carried out in the context of the Master in Informatics Engineering (MEI) with specialization in software engineering at the University of Coimbra. Thus, this dissertation should help research in the fault injection field and allow the creation of new research strands to study new approaches. It is also integrated in VALU3S [1], a European project that aims to evaluate the state-of-the-art V&V (verification and validation) methods for automated systems in the industries such as agriculture, railway, healthcare, and aerospace.

1.2 Goals

Our main goal is to develop failure model injection (also referred to, in this dissertation, as failure injection or the injection of failure models), compare it against traditional fault injection and validate its ability as a technique capable of accelerating the FI process compared to traditional FI techniques. Although this type of injection has already been used in other contexts, we aim to validate in which situations it can replace FI and which failure models and injection points generate more realistic results.

The other objective was the integration of our approach in the ucXception framework in order to provide free access to the approach and enable other researchers to use it and help in new research works that target the injection of failure models.

1.3 Contributions

As a result of this dissertation, we have written a paper submitted to ISSRE 2023 Conference [2]. This paper summarizes the work conducted in this dissertation, namely, the experiments and results obtained from the comparison between FI and failure model injection.

We also contributed with a failure model injection tool in ucXception [3] so that other people can use the tool for various academic purposes and new studies involving failure models.

Lastly, we have created a GitHub repository [4] with all scripts necessary to perform campaigns in a virtualized setup identical to ours so that it can be used in future works of FI or failure model injection field. It also comprises the scripts for the result analysis used in this dissertation.

1.4 Document Structure

This dissertation comprises seven chapters: Introduction, Background, State of the art, Approach, Results, Methodology, and Conclusion.

[1]https://valu3s.eu/
[2]https://issre.github.io/2023/
[3]https://ucxception.dei.uc.pt/
[4]https://github.com/ucx-code/fi-campaign-manager

The Chapter 1 identifies the scope and context of the dissertation and the report's motivation, goals, contributions, and structure.

In Chapter 2, a deep study of the multiple topics addressing this dissertation and that support our investigation is presented. Firstly, the topic of dependability and its various means are explained. Dependability refers to the evaluation of a system and its service correctness. Afterward, we detail fault injection, a more specific technique of dependability, and the main topic of this dissertation. This section presents the multiple fault injection classes as well as the existing tools for FI experiment performance.

The Chapter 3 describes in more detail the ucXception tool since it is the framework where we will integrate our approach. Then we present FI acceleration topic and its multiples techniques for hardware and software faults. Finally, we present an overview of some existing failure injection techniques to accelerate FI experiments and some tools that companies use to test their systems through injection of failures.

The fourth chapter describes the setup that was used for the validation of our technique. We also present the main characteristics of our approach, such as the possible targets for the injection, the type of failures we injected, and the metrics that were used to compare our technique with the traditional FI.

The Chapter 5 presents the results of our experiments. We studied each defined metric and compared the data resulting from our approach with the traditional FI datasets. In the end of the analysis, we presented a summary of all the results. We also suggest future work that can be developed from the results and conclusions we have achieved with this study.

Chapter 6 presents the plan and methodology for the first and second semesters by describing the multiple stages of work and the respective Gantt charts. This section also defines the limitations we faced during the dissertation, the risks associated with the work developed in the second semester, and the mitigation plans for each.

Lastly, the Chapter 7 summarizes the research scope and the conclusions we retrieved from all the work developed during the year.

Chapter 2

Background

This chapter provides all the background concepts of dependability in software systems, an overview of fault injection, and its multiple tools that can be used for fault injection campaigns.

2.1 Dependability

Nowadays, different systems are required to be trustworthy due to their increasing use in multiple services that can be critical. Therefore, they cannot fail and must always be available, or at least most of the time. Examples of services that can be critical are bank management, hospital monitoring, or even airplanes. Software should deliver correct service, that is, do the functions it was designed to and implemented. Thus, dependability is an important factor that should be considered when evaluating a specific system.

A possible definition of dependability is "the ability to deliver service that can justifiably be trusted" [2]. However, this definition is vague because it raises the question: how do we consider a system to be trusted? Furthermore, trust is subjective, it cannot be measured and may vary among different people and systems.

Therefore Avizienis et al. give another definition: "dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable" [2]. This idea means that during the execution of a service, the system is probable to fail at some point, and we can define a level of acceptance for the service failures that should not be exceeded. Dependability aims to control failures and minimize its damage to the system in question.

2.1.1 Dependability Attributes

Dependability encompasses multiple attributes that should be used to characterize and validate a system. If it meets those attributes, then it can be trusted. As Avizienis et al. [2] refers, these attributes are availability, reliability, safety, integrity, and maintainability. Although these attributes may be confused and overlap, there are differences between them and the way they relate to the system.

- **Availability** refers to the system being operable most of the time. Modularity and Redundancy can help a system reach the pretended level of availability [3].
- **Reliability** is the system's ability to keep providing the correct service over time.
- **Safety** is when the system might cause catastrophic events that may cause death or environmental disasters. This attribute might not be present in every system since not all systems evolve this type of danger when they fail or stop.
- **Integrity** directs that unauthorized people or third-party systems should not alter the respective service without authorization.
- **Maintainability** means that when a system fails, its repair or maintainability should be easy.

Availability and reliability are two terms that seem similar, although availability only concerns being operable all the time. On the other hand, reliability is the deliverance of service correctness. Moreover, they have different ways of being calculated. Reliability is proportional to mean time between failure (MTBF) [3], which is the interval time between a system failure, an irregularity in the execution, and the next one. The formula to calculate the reliability of a system is $R(t) = e^{-\lambda t}$ where $R(t)$ represents the reliability of the system at time t and λ is the failure rate [4]. While availability results from the measure of two mean value times: MTBF and mean time to repair (MTTR), meaning the mean time it takes to repair from a failure. Thus, the following formula estimates the exact percentage of availability $\frac{MTBF}{MTBF+MTTR}$

2.1.2 Fault

The dependability of a system is susceptible to threats. The first threat to dependability is the fault, which can be a bug in the software code or a defect in one of the multiple components composing a system, thus leaving the system in a state of fault where it is vulnerable to errors. When a fault is present but is not activated, it is described as **dormant**. Faults can be classified with the help of classes, and each can belong to more than one type depending on its characteristics. Table 2.1 presents a scheme with all the fault type classifications and its description.

Viewpoint	Class	Description
Phase of creation or occurrence	Development Fault	Occurs in the development or maintenance phase
	Operational Fault	Happens when a user is using the service
System Boundaries	Internal Fault	Source is in the product itself
	External Fault	Results from the user or environment
Phenomenological cause	Human-made Fault	Human actions caused it
	Natural Fault	Its causes did not have human participation in it
Dimension	Hardware Fault	Affects hardware
	Software Fault	Affects programs or their data
Objective	Malicious Fault	Introduced with the intent of damaging the system
	Non-malicious Fault	There is no intention to harm the system
Intent	Deliberate Fault	Created due to an erroneous decision
	Non Deliberate Fault	Unintentionally introduced
Capability	Accidental Fault	Introduced inadvertently
	Incompetence Fault	Lack of professional competence
Persistence	Permanent Fault	Persists in time, even if we restart the system
	Transient Fault	Determined in time and may disappear with a restart

Table 2.1: Fault Classification Table [2]

One fault can belong to multiple classes, although some cannot be used together in a fault classification. For example, a fault cannot be classified as both natural and intent simultaneously since a natural fault has no human intervention. Thus, we cannot classify the purpose of its introduction into the system.

As mentioned in Table 2.1, a fault can be permanent or transient depending on its persistence in the system over time. Within the permanent faults, we can distinguish them according to their reproducibility. If a fault's activation is reproducible, it is a **solid** or **hard** fault, whereas if it is not reproducible, it is defined as an **elusive** or **soft** fault.

Transient faults can occur due to energetic particles. For example, a cosmic ray can strike an electric circuit and cause a soft error which can be translated into a bit-flip or the inversion of a state of a given processor [5]. Although such situations are rare, they can happen in all systems, particularly those farther away from the earth's surface (e.g., aircrafts, satellites), as these are more subject to cosmic rays.

Elusive and transient faults can be grouped due to the way that they manifest as intermittent faults [2], which is a fault that occurs intermittently, i.e., disappears and shows up randomly, usually caused by an unstable component in a hardware system [6].

2.1.3 Error

A fault activation causes an error. That is, the fault passes from a dormant to an active state which can cause the incorrect service of the affected component and, ultimately, may lead to the system's failure. If the service detects the error and activates an error message, it is defined as **detected**. By contrast, if the system does not detect it, although it is present, it is called **latent error** [2]. Typically an error affects one component and is called a **single error**, but if it involves more than one component, it is called **multiple related errors**.

In a complex system, an error is even more harmful due to the probability of error propagation which is when an error turns into other errors, thus causing an internal propagation within the component. When a component is dependent on another component, it can propagate the error to it, leading to its failure, thus providing incorrect service. Nowadays, systems are becoming more complex, involving multiple components. Therefore, an error in a single component can affect all of them through a propagation chain that can lead to a system failure.

2.1.4 Failure

A failure scenario is when a system deviates from the foreseen outcome. In other words, the user or component that depends on the affected service detects an anomaly that prevents it from performing its expected execution. Thus the service cannot meet the requisites for which it was designed. A system does not fail the same way every time and thus can have different failure modes, where each one might have a different degree of severity too. Therefore, it is possible to characterize an incorrect service according to four service failure mode viewpoints: domain, detectability, consistency, and consequences [2].

The first viewpoint, the failure domain, refers to what failed in the system. The failure can be a **content failure**, meaning that the service content presented to the user differs from the expected, or a **timing failure**, when the content might be accurate but delivered out of time, too early, or late compared to the desired time. However, a service can fail simultaneously in content and time. In that scenario, it is possible to distinguish the failure in two modes: when the system blocks and no activity is observable by the user, which is a **halt failure**. Although, if the system delivers its service but is inconsistent, it is an **erratic failure**, and therefore the service cannot be trusted.

Another viewpoint is detectability. A system should check and validate the delivered service's correctness through detection mechanisms to avoid a failure going unnoticed during service use. When the system signals a deviation from the correct service, the failure is defined as a **signaled failure**. In reverse, if the failure goes unrecognized, it is an **unsignaled failure**.

Consistency is about how different users of the service notice the incorrect service of it. So, a service failure can affect all users in the same way, which is a **consistent failure** since it is general to all. However, if only some users perceive a similar incorrect service, it is an **inconsistent failure**. In that case, some may experience correct service usage and not notice the respective failure.

To complete the four viewpoints used to characterize a failure is the consequences of it. The various consequences of a failure must be taken into account and scrutinized in order to able to prevent undesirable and dangerous situations for the user and for those who provide the system. If a failure causes insignificant damage to the system and its service, it is a **minor failure** because its losses will be small or nonexistent compared to the gains of the correct service. Thus, there is no great urgency to resolve it. Although, if the failure endangers human lives or its losses are enormous compared to the gains that the correct service generates, it is classified as a **catastrophic failure** and presents an urgent character at the level of resolution to avoid the mentioned above situations. These viewpoints support the characterization of a failure so that its mitigation can be adjusted and it is possible to have adequate control over failures in different situations.

As mentioned, a fault can originate a soft error that can generate failures if there is no protection against it. **Silent Data Corruption (SDC)** is the most severe type of failure that can arise and refers to errors that are not detected by fault detection mechanisms, thus affecting the outcome of the system. If the error is detected, although not rectified, it is called a **Detected Unrecoverable Error (DUE)**. When the outcome was not affected by the failure, then it is defined as a "false DUE", but if the failure affects the outcome, then it is a "true DUE" [5].

There are multiple studies addressing the origin of failures since it is vital to guarantee that a system delivers the correct service consistently. For example, Jim Gray studied the various system failure modes of Tandem Systems [3], the dominant enterprise of fault-tolerant computer manufacture. He concluded that the administration of the service, such as operator action, system configuration, or maintenance and software, are the dominant sources of failures. On the other hand, hardware and environment (fire, flooding, etc.) are the minor cause. Oppenheimer et al. analyzed three large-scale internet services, each with different features and purposes, an online service, a content hosting service, and a read-mostly one [7]. They found that operator error is the most significant failure cause of two services (online and content) and software errors that affect the network for the read-mostly service.

2.1.5 The Chain of a System Failure

In summary, a service can enter an erroneous state due to a fault, which can cause it to not fulfill the expected functional or non-functional requirements, hence leading to a service failure. Furthermore, systems are becoming more complex and extensive due to the evolution of technologies and their uses, thus it becomes more common for a system to suffer from faults that may affect its execution.

Figure 2.1 schematizes the events that may lead to a system failure. In this example, the target system comprises components A and B. When a present fault is activated, it will generate an error in the respective component that may cause it to fail. In this situation, component B is dependent on component A. Therefore, incorrect service delivery from component A will affect component B, which may also generate a fault that becomes an error and later a failure. Thus, component B fails, and because it is crucial for the correct service of the target system, it will lead to the same sequence of fault, error, and failure propagating to the main system, and thus the system can become inoperable. The chain of a system's failure works the same way for most systems composed of multiple components. It is, therefore, essential to mitigate fault occurrences and their effects before they propagate.

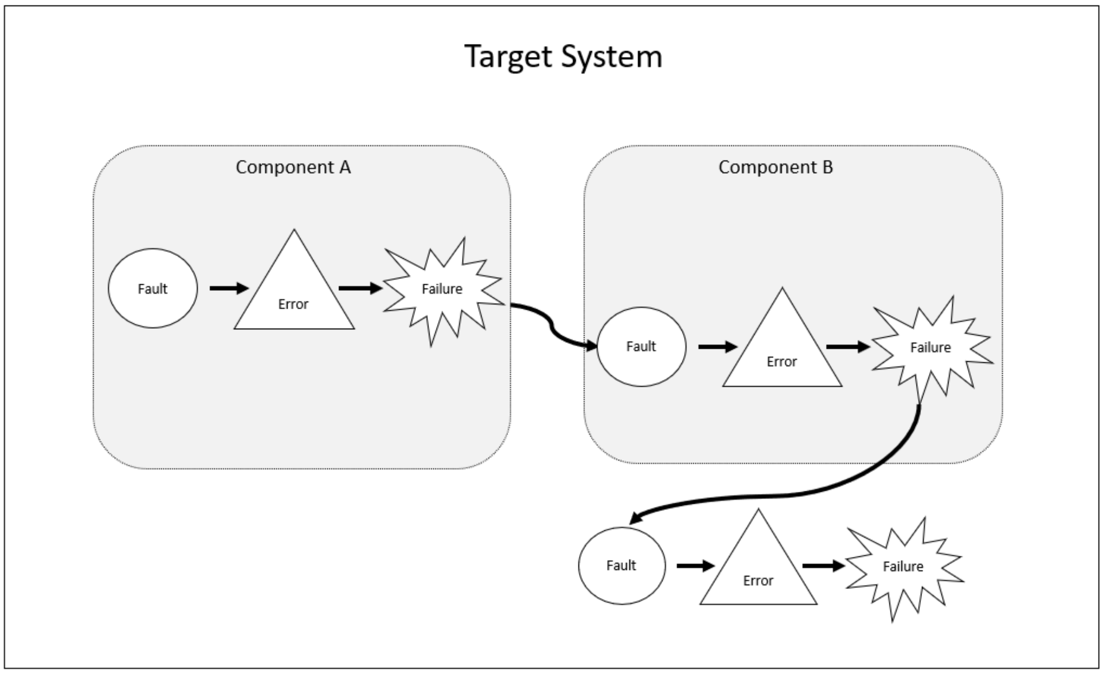


Figure 2.1: The chain of a system failure [2]

2.1.6 Dependability Means

As explained above, as the complexity and the number of components that compose a system increase, so does the probability of service failure. Some techniques can help improve dependability by acting on the faults that a system may contain. These techniques can be divided into four categories with the same goals but different approaches for achieving dependability. They are fault prevention, fault tolerance, fault removal, and fault forecasting.

Fault prevention is a set of techniques that tries to prevent the happening of a fault in order to deliver a reliable service. Although this technique may go unnoticed, it is used in most development projects through methodologies and rules common to all teams. The management of a project is a critical area for its success. For example, in the requirement phase, some incomplete or ambiguous requirements may lead to potential faults in the development phase. Thus it is crucial to have adequate management to prevent faults from happening.

Another category is **fault tolerance**. These approaches try to make the system maintain its service correctness in the presence of fault through tolerance of it. Techniques such as fault-masking [2] or n-version programming [8] belong to this category.

If the technique aims to remove the fault from the system, it belongs to the **fault removal** category. Common techniques to remove faults are model checking or simply testing the system to verify its functions.

The last category is **fault forecasting**, which contains techniques that aim to predict the number of faults and their incidence. Currently, these techniques are not widely explored in terms of fault field.

More than ever, computer systems need to be resilient, i.e., keep executing in the presence of faults or undesirable situations, since they are increasingly used and require involving a great complexity, thus being impossible to be free of vulnerabilities. Hence it is common for systems to contain fault handling mechanisms in order to contain faults and thus increase their resilience. However, these mechanisms must be tested in multiple scenarios and with multiple types of faults. Because of this need, the fault injection technique arises.

2.2 Fault Injection

Fault Injection (FI) is an experimental set of techniques that consists of intentionally injecting or emulating faults into a system computer to study how it will behave in its presence. R. Barbosa et al. define FI as "the process of deliberately introducing faults or errors in computer systems, allowing researchers and system designers to study how computer systems react and behave in the presence of faults" [9]. This means that it allows evaluating and analyzing how a system handles a fault and its effect.

Faults are injected into a system defined as the target system, which is also the system to be analyzed and evaluated. For that, the system must perform a sequence of predefined tasks to behave as in a typical execution of the system. This sequence is called **workload**. In contrast, the set of faults injected in the target system is called **faultload** [9].

A **fault injection experiment** injects a single fault in a target system and analyzes the system and its behavior. Typically, for an accurate analysis of the system, multiple faults are injected so that it is possible to analyze the behavior of the system in many situations. The injection of a set of faults is called a **fault injection campaign** [9].

It is used to evaluate and validate fault handling mechanisms. It can also be used to analyze the impact of various faults and system behavior in their presence and for dependability benchmarking.

2.2.1 Fault Injection Properties

There are many different techniques for fault injection, and each one has different characteristics that should be compared to measure the viability of each technique and simplify the choice of a particular technique to apply. Therefore, a set of properties characterize every FI technique [9][1]:

- **Controllability** is controlling the time and location of the injection of faults.
- **Repeatability** is the ability to repeat the same experiment multiple times and get the same result.

- **Observability** is the ability to record and see the FI experiments to analyze their effects.
- **Reproducibility** means the capacity to reproduce the result of a FI experiment.
- **Representativeness** is the accuracy of a workload, a faultload, or the target system compared to reality.

These five properties can apply to every (FI) experiment, although other properties can be considered in some specific injections [10]:

- **Reachability** is the ability to reach the fault location.
- **Intrusiveness** is the unintentional impact on the temporal and spatial behavior of the target system.

There are two modes of testing a target system, 1) injecting faults on a real system or 2) injecting faults in a model that simulates a software system or emulates a hardware system. We have a more accurate experiment in real systems since it is the system itself that we want to test, i.e., higher system representativeness. Furthermore, when the goal is to test fault handling mechanisms, it is more authentic since we are testing the actual mechanisms [9].

Fault models used in simulation or emulation based systems may have higher fault representativeness than injecting artificial faults on a real system since it is not so accurate because we have no guarantees that the real faults are similar to those injected. Also, the properties referred to above are higher in simulation mode since there is better control over the target system and the fault injected. However, it always depends on the simulation model's degree of accuracy [11]. Finally, although there are some drawbacks regarding the overhead, in hardware emulation and especially in software simulation, there is a significant time overhead [6]. Currently, multiple fault injection techniques use both ways to test. These techniques are called Hybrid Fault Injection [9].

The application of FI requires much complexity since, for an accurate analysis of the system, it is also necessary that the faults are precise and representative of reality. Without a good representation of reality, a system cannot be guaranteed to behave as it does when there is a real fault. Therefore three parameters should be considered and studied when we want to perform a FI experiment to ensure the injection has accurate results [12][13].

The first is **fault type**, which can be seen as "what to inject", meaning the fault model and its characteristics. This choice defines the fault representativeness which should be as high as possible so that the FI experiment is as close as possible to the reality we want to evaluate. Another parameter is **fault location** which means "where to inject", the location on injection is essential because it is not viable to inject a fault in every location since there are a massive number of locations where we can inject [1]. Lastly is the **injection time** meaning "when to inject", which needs to be considered to achieve a good accuracy of the time when the fault happened in a realistic situation since the execution of a system has multiple states that vary across the time.

It is possible to inject various types of faults into multiple systems. The following sections present the different hardware and software FI classes.

2.2.2 Hardware Fault Injection

Hardware Fault Injection (HFI) is the injection of faults regarding the hardware part of a system, for example, a component or a microcontroller. These faults can occur due to production defects, the aging of the components, or from radiation [14].

A hardware fault can be permanent, transient, or intermittent [6]. Depending on the injection injection form that is chosen, these three types can be injected with more or less difficulty.

Hardware Implemented FI

Hardware Implemented Fault Injection (HWIFI) includes techniques for injecting physical faults in the target system using hardware. Although it is not the most used technique type, it has some advantages in injecting via hardware. One advantage is that for the injection of a fault is not necessary to change the target hardware, i.e., the intrusiveness is minimum for most of the techniques inserted in this category. The fault representativeness is expected to be higher since actual faults are injected into the real hardware [9]. Also, the reachability property is typically higher since it is possible to reach specific fault locations with some techniques involving radiation [6]. On the other hand, there are some drawbacks like the cost. The application of fault injection experiments through hardware may need particular types of equipment that can be expensive [15]. Furthermore, these techniques usually have lower controllability because it is difficult to manage when a fault is going to happen and the system's state. Likewise, these techniques have low observability since the data collection to analyze the results may be complex, making the technique not viable for some experiments [6] [16].

One technique is the **pin-level** FI. This technique uses pins of digital circuits, changing their logical values from 0 to 1 and vice versa, simulating a "stuck-at" fault through probes connected to the circuits [17]. However, the technique became outdated due to the increasing complexity of the electronic circuits of most available computer systems [9]. **Power supply disturbances** (PSDs) are infrequently used due to low repeatability. However, they can complement other FI techniques for evaluating error detection mechanisms [9].

Another technique used is **test port-based**, which consists of setting a breakpoint to know the target location and then reading the target location's value via the test port, manipulating it, and writing it back with the fault value. This technique is suitable for modern microprocessors since they are built with debugging and testing features accessed with test ports. The fault type to be injected depends on these features supported by microprocessors [9].The principal advantage of this technique is the minimum intrusiveness that it causes to the target system. However, this injection type has some limitations, such as the difficulty of accessing hardware structures in the microarchitecture that are invisible to the programmer since the manufacturers usually do not disclose information accessing that structures. The time overhead can impact the efficiency of FI, and ports such as JTAG or BDM are low-speed ports which can be a hindrance in the experiences [9].

Radiation-Based FI

As said before, radiation can hit a system component and make a bit-flip in its circuits. This bit-flip can be defined as a Single Event Upset (SEU), a transient fault that affects registers and microcontrollers' memories [14]. Therefore, a way of injecting faults into a target system is by exposing it to radiation to evaluate fault tolerance mechanisms [9]. Furthermore, this type of injection allows better reachability since it is possible to inject faults in challenging accessible fault locations. For example, with the **Heavy-ion** method, it is feasible to inject faults in VLSI circuits in exclusive locations [6]. However, the controllability of this set of techniques is very low since we cannot control the time and location of a specific injection, leading to low repeatability [9].

Software-Implemented Fault Injection of Hardware Faults

One manner of injecting faults in target hardware is through software. This set of techniques is called Software-Implemented Fault Injection (SWIFI) and consists in executing a software code that emulates a hardware fault in a specific component of the target system, like the register or memory corruption [18]. This type of injection has been increasingly used since it presents less complexity and cost than hardware-implemented FI. Another advantage of this set of methods is their portability because it is more readily applicable to other fault types and systems. However, there are some limitations, like the difficulty of emulating a permanent fault since it needs to manipulate that fault every time the system reads it. Their reachability is usually lower than hardware-implemented [18]. As is also its fault representativeness since faults generated by software may not be similar to actual physical faults [17]. As R. Barbosa et al. says, the representativeness depends on the presumption that SWIFI can emulate real hardware fault effects [9].

There are two typical approaches for hardware faults emulation by software: **run-time injection**, where faults are injected during the system's workload execution. In this approach, the run-time overhead may be significant since it needs to stop the workload to inject a fault and then start the workload again, thus being more intrusive to the target system. The other approach is **pre run-time injection**, and this technique presupposes faults are injected before the workload begins, having less run-time overhead in comparison with run-time approaches. However, performing a complete fault injection campaign is more time-consuming since it needs to prepare each fault injection that composes a campaign [9][17].

Simulation-Based Fault Injection

FI can be conducted through the simulation model of the target hardware in different levels like the device level, logical level, Instruction Set Architecture (ISA) level, block level, or system level [9]. The target system and faults are modeled in software, typically a fault simulator [16].

A critical point of simulation FI is system representativeness since the simulated model needs to be accurate with the real one so that the FI results may be representative of reality [6][14].

This type of injection allows the injection of both permanent and transient faults. Usually, it presents higher controllability due to better control of a fault's timing and location and high observability, thus more appropriate for rigorous fault injection effects analysis [18]. Also, the level of intrusiveness is minimum since the injection is not on the real system, not having the risk of damaging it. Therefore, it is a good way of evaluating a system in the early stages of development and design when there is no physical prototype [14][17]. Although there are some drawbacks, like the development efforts it takes to model the target system and faults model, the accuracy of the model. Also, it is very time-consuming the performance of simulation tests [6][16].

Hardware Emulation-Based Fault Injection

As referred, simulation-based is very time-consuming in performing the FI experiments. Therefore, Hardware Emulation-Based FI arises to overcome this limitation [16]. This technique uses Field Programmable Gate Arrays circuit for the hardware emulation to speed up the experiments [6]. This technique has the same benefits as the simulation-based and is less time-consuming for the experiments. However, the observability property can be low since communication with the emulated circuit is needed to collect and analyze the results [9].

Hybrid Approaches for Hardware Fault Injection

Nowadays, it is common to combine different techniques of injection in order to improve the accuracy and efficiency of a FI campaign. A typical combination is the software-implemented FI with monitorization by hardware which is usually more accurate [6]. These combinations are made to extract maximum efficiency and consistency of results from a fault injection.

2.2.3 Software Fault Injection

Software faults are one of the principal causes of failures in a system due to the common mistakes that programmers make in developing a system [3][7]. Therefore, the system must handle its respective faults and maintain its resilience. This situation is tested and validated through FI of software faults.

Software Fault Injection (SFI) is recent in comparison with hardware FI [12], and the fault target is software, thus being difficult to inject software faults through hardware. Therefore the techniques for SFI use only software-implemented fault injection [9].

However, a critical point of SFI is fault representativeness since it is difficult to predict the mistakes made by a human, and modeling it is incredibly complicated and subjective [9] [19]. For an accurate FI experiment, the injected faults must reproduce the residual faults of a program. Residual faults are the ones present in the software that were not detected by the testing of the system [12].

SFI has two main approaches that can be used to perform an experiment in a target system: fault injection and error injection.

SFI with Error Injection

Error injection tries to emulate the effects of a fault by injecting the error that a fault can generate, i.e. instead of injecting the software faults, we emulate the effects of it [12]. According to R. Barbosa et al., two main techniques are used when injecting errors: **program state manipulation**, which consists of altering variables, points, and other data stored in CPU-Registers or main memory. In comparison, **parameter corruption** is modifying functions parameters, procedures, and system calls [9].

Error injection is a helpful method for robustness testing since representativeness is not the critical point for that type of testing because the goal is to discover frailties in the component that builds a system [12].

FI of Software Faults

The other way to evaluate a system concerning software faults is by changing the source code, object, or machine code to introduce a fault. These manipulations are defined as **mutations** [9].

The characterization and classification of software faults are crucial for an accurate and consistent FI experiment. Thus, Durães et al. created a fault model based on analyzing several programs where they collected the faults found in them [19]. They used 12 programs as fault sources and found 668 faults in total. First, they classified the faults according to the Orthogonal Defect Classification (ODC), a technique for fault classification that is widely used and allows comparison with other techniques. This method categorizes a fault according to its characteristics and can be an assignment, checking, interface, timing/serialization, algorithm, or function. Then, through precise analysis and study, they defined thirteen operators, each of which contemplates a type of fault [19]. Table 2.2 presents a table with the defined operators.

This fault model is one of the most accepted and, therefore, one of the most used in SFI [12][20]. Although representativeness remains a critical point, it is still studied to improve the accuracy of SFI.

Operator	Fault Type
OMFC	MFC - Missing Function call
OMVIV	MCIC - Missing variable initialization using a value
OMVAV	MVAV - Missing variable assignment using a value
OMVAE	MVAE - Missing variable assignment using an expression
OMIA	MIA - Missing if construct arround statements
OMIFS	MIFS - Missing if construct plus statements
OMIEB	MIEB - Missing if construct plus statemnets plus else before statements
OMLAC	MLAC - Missing "AND EXPR" in expression used as branch condition
OMLOC	MLOC - Missing "OR EXPR" in expression used as branch condition
OMLPA	MLPA - Missing small and localized part of the algorithm
OWVAV	WVAV - Wrong value assigned to variable
OWPFV	WPFV - Wrong variable used in parameter of function call
OWAEP	WAEP - Wrong arithmetic expression in parameter of a function call

Table 2.2: Defined Fault Model [19]

If we are concerned about the accuracy of the effects caused by the injected faults, then this approach is suitable compared to error injection since the goal is to inject faults that are representative of the ones existing in the field of a target system. However, this goal is not easy to achieve since it is necessary for a thorough analysis of the location of the fault, i.e. which instructions should be changed to represent the actual faults in a system [12]. For example, in a random location, SFI campaign will have fallacious results since we are injecting faults in locations that might never activate the respective fault, and this process can be slow. A drawback of mutations is that they involve access to the source code and hence a recompilation of code which can be a long process. Durães et al. developed the G-SWFIT technique [19], which consists in performing the mutations on the binary code. However, this technique raises problems in the mapping process between the fault model and the machine instructions.

In order to inject hardware or software faults into a system, a set of tools can be used, and they provide different techniques and characteristics more suitable for specific situations. The following section briefly presents multiple tools for FI.

2.2.4 Fault Injection Tools

Multiple tools are used for injecting faults and further analysis of the campaign. These tools vary in their fault model and the injection process. Therefore, each tool provides a set of properties, making them better for a determined FI campaign. Next, two commonly used tools are presented, GOOFI-2 [10], and Xception [18], each with specific features and techniques. We detailed these tools more because they are widely used and present different ways of injecting faults.

One widely used tool is **GOOFI-2**. This tool is used for FI of hardware faults and provides three techniques. One is a test port-based technique defined as **Nexus-based injection**, and it injects faults via the Nexus ports through a set breakpoint to inject the fault. It provides high observability and reachability. To perform Nexus-based, it is not necessary to change the workload making the spatial intrusiveness low. However, its temporal intrusiveness is high. The second is a SWIFI technique named **Exception-based FI** and uses exceptions activated by hardware breakpoints to inject faults. It provides high controllability. However, compared with Nexus-based, the observability is low. The last one is the **Instrumentation-based FI** a SWIFI technique, where faults are injected by creating a software breakpoint and expanding the workload with an instrumentation function. In this technique, the spatial intrusiveness is low and has high repeatability and controllability. Although it does not support the target system's monitoring, the Nexus debugger can overcome the drawback. The fault model of GOOFI-2 is the emulation of transient hardware faults affecting registers and memory of a microcontroller using single or multiple bit-flip errors [10].

Xception is another tool for FI and provides SWIFI techniques. Nowadays, most modern processors contain advanced debugging and monitoring features, and Xception uses them for fault injection. In other words, Xception injects faults through the raised exceptions, making it the fault trigger during the target program execution. Furthermore, it can monitor their impact through the monitorization features of the processors. Since it uses exceptions of the program, the interference with the target program is minimal since it is not modified. The fault model of Xception is transient hardware faults like stuck-at-zero, stuck-at-one, and bit-flip on memory [18] [9].

Table 2.3 presents other FI tools that can be used for FI experiments. However, it is worth noting that tools like FIAT, FERRARI, FTAPE, DOCTOR, and Xception can potentially be used for software faults since they manipulate the system state, although there is no support for that.

Tool	Tool Description	Fault Model
MESSALINE [6]	Used for pin-level FI using both active probes and sockets	stuck-at and complex logical faults
FIAT [9]	Corruption of the code or the data area of the program's memory	zero-a-byte, set-a-byte and two-bit compensation
FERRARI [9]	It uses software traps to inject CPU, Memory, and bus faults.	address line, data line and condition code faults
DOCTOR [15]	Used for injection in distributed real-time systems	memory faults, CPU faults and communication faults
FTAPE [9]	Used to evaluate benchmarking and computers of fault tolerance	memory faults, CPU faults and I/O faults
DEFINE [9]	Mutation of assembly language listings of the target program.	Software Faults: initializations, assignment, condition check and function faults
G-SWFIT [19]	Perform mutations at the machine-code level in order to emulate software faults	The used fault model is the set of operators defined by Durães et al.

Table 2.3: FI Tools

Chapter 3

State of the Art

The state-of-the-art chapter contains various approaches to accelerating injection experiences. These techniques are divided into FI acceleration techniques for hardware and software faults. Then it presented some used failure injection techniques to accelerate experiments and some tools used by companies.

3.1 ucXception Tool

ucXception [1] is an open-source framework developed at the University of Coimbra and allows performing FI campaigns. Furthermore, it supports the injection of hardware and software faults, and the experiments can target different systems, such as virtualized and cloud computing systems.
One of the advantages of ucXception is that it is not necessary to have deep knowledge about FI and its specifications because the use of the tool is through a graphical interface that the framework provides, thus being easy to use. The other advantage is the ability to integrate new tools in ucXception, making it expandable to new experimental tools and techniques [21].

3.1.1 ucXception Architecture

The architecture of ucXception is mainly composed of two modules, frontend, and backend.

The frontend is the graphical interface provided to the user where he can manage his campaigns and evaluate them. More specifically, the user can execute preliminary analysis, create new campaigns and analyze and download its results. As mentioned, the interface is easy to use, so the framework is suitable for any user with minimal knowledge of FI [21].

The backend module comprises two components: the Manager and a REST API. For each campaign in execution, there is an instance of the Manager, which is the

[1] https://ucxception.dei.uc.pt/

main element of the ucXception framework since it is responsible for the execution of FI campaigns and storing results. The results are stored in CSV files in an SQLite database that also contains the user information. Lastly, the REST API is responsible for the connection between the Manager and the graphical interface, the frontend, through HTTP requests.

Figure 3.1 presents the architecture designed by P. Almeida et al. of ucXception [21].

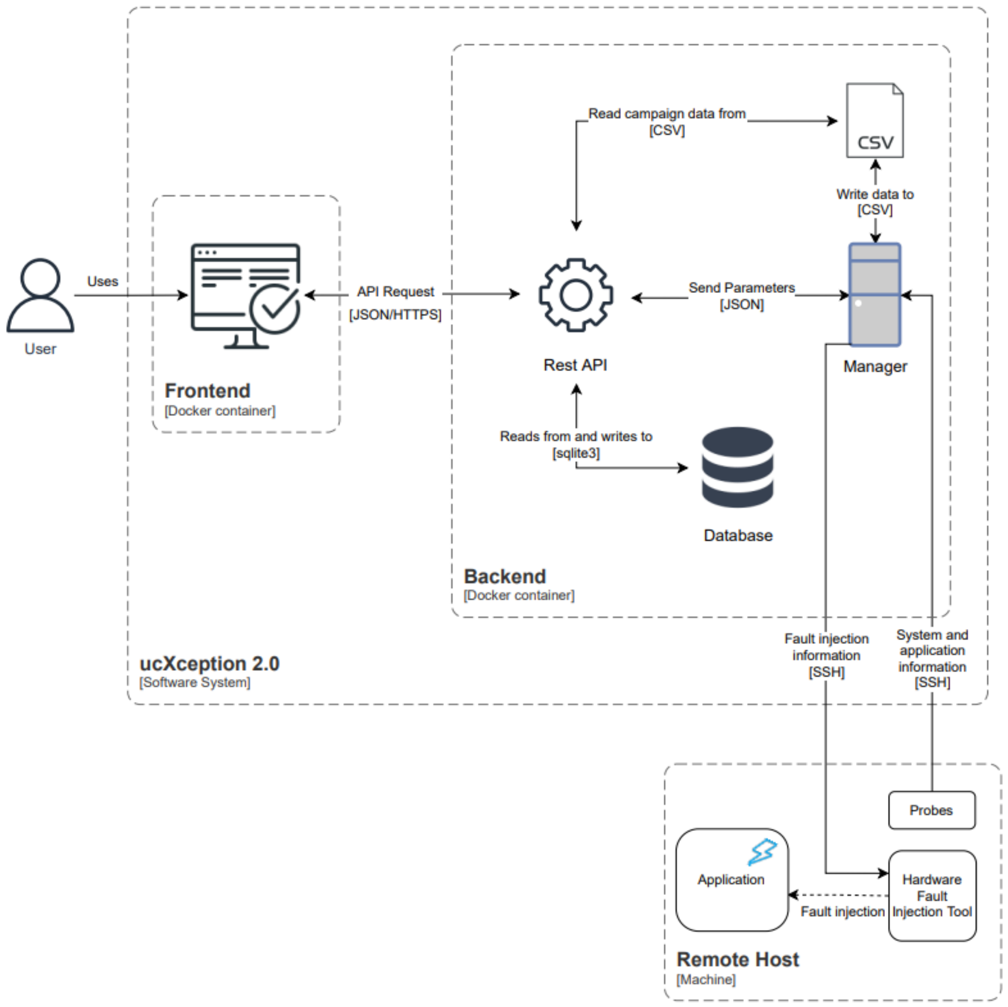


Figure 3.1: ucXception architecture [21]

3.1.2 ucXception Components

ucXception has several pre-made components that the user may use in its campaigns to monitor and evaluate the target system. Although a variety of components already exist, the user can add new ones to the framework and thus increase the accuracy and efficiency of FI campaigns. The components are divided into pre-probes, post-probes, parsers, and transformers [21].

Pre-probes collect system-wide metrics, thus being used to monitorization of the target system and collect metrics that can later be used for system analysis when running the fault injection campaign. Various pre-probes are already available in the framework: Logs probe, Intel PCM probe, Ping probe, SAR probe, TCPDump probe, and Xentrance probe, each with specific metrics monitorization.

Post-probes are used to monitor specific processes instead of all systems like pre-probes. Currently, only one post-probe is available, the Pidstat probe.

The **parsers** convert the results from a FI campaign to a more practical and compact format, thus being helpful for the later analysis of results which are always written in CSV files. The available ones are HW FI parser, SW FI parser, Pcap -> TCP parser, Info parser, MD5 output parser, Return code parser, and Current Folder parser.

The last component category is the **transformers**, which are identical to parsers, but the output generated by transformers is stored in individual files in their own results folder. They are mainly used to convert the output of the probes into a more effortless format. The available ones are: Pcap -> TCP 2 CSV transformers, Pidstat 2 CSV transformer, SAR 2 CSV transformer, Ping 2 CSV transformer, and save output transformer.

More information and detail about each component is present in P. Almeida et al. article [21].

3.1.3 ucXception Fault Injectors

ucXception framework provides three default fault injectors that implement different fault models. Two of these are used for hardware fault emulation, and the other for emulating software faults. Although only three fault injectors are available, as mentioned, ucXception is expandable, so it is possible to integrate new fault injectors.

One of the fault injectors is **Hardware faults in Linux-based systems**. This injector emulates soft errors, namely bit-flips, in CPU components and registers. Although it is only possible to perform bit-flips in CPU registers, there is no support for executing them in memory since it is possible to accurately represent this injection type with injection in the register files. Firstly, the fault injector attaches to a process through the aim of ptrace functionality available in most Linux installations. Then, collect the register data structure, perform the bit-flip, and resume the program target system execution where the registers have suffered from a bit-flip. This tool performs SWIFI, so no changes are needed in the target system's hardware. The tool also provides logging functionalities for a more detailed analysis of the respective campaign [21].

Another FI tool available is **Hardware faults in virtualized systems**. As the name says, this tool is used for virtualized systems. Like the previous one, this tool injects bit-flips, which can target CPU registers. The injection involves changing the register value kept in the data structure that stores a Virtual Machine's (VM) CPU state that is updated immediately before a context switch. Although, this technique is dependent on the rate of context switch occurrences. This switch rate must be accurate and analyzed since the bigger it is, the higher the precision of the injection, however if it is too elevated, it will cause a considerable overhead and intrusiveness to the target system [21]. Figure 3.2 presents this tool FI scheme.

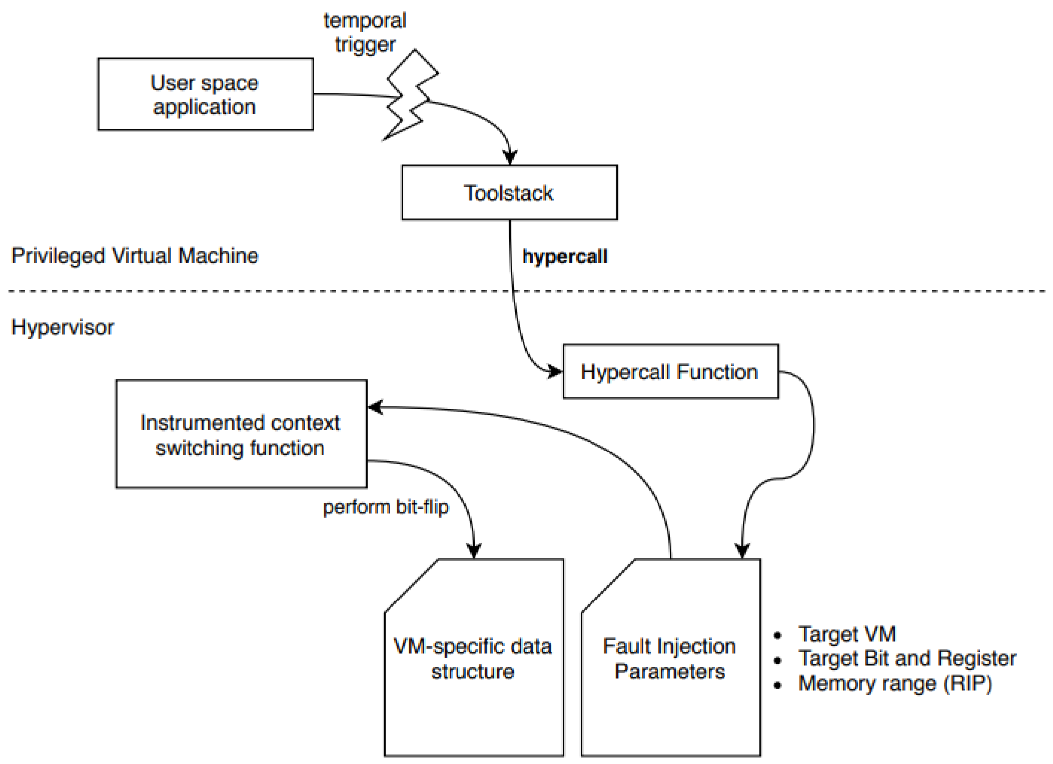

temporal trigger
User space application
Toolstack
Privileged Virtual Machine
hypercall
Hypervisor
Hypercall Function
Instrumented context switching function
perform bit-flip
VM-specific data structure
Fault Injection Parameters
• Target VM
• Target Bit and Register
• Memory range (RIP)

Figure 3.2: ucXception - Hardware Faults in Virtualized systems FI [21]

The FI process starts on the privileged virtual machine (PVM), known as dom0 in Xen's nomenclature, where the ucXception activates the trigger functionality. When the trigger is activated, the toolstack is called and performs a hypercall to a hypervisor function and sends the FI parameters. The parameters for the FI are the target Virtual Machine (VM) since the system may have multiple VMs, and the goal is focused on one. Also, the target register and target bit and the last parameter, which is the start and end memory's range which the *rip* register will be pointing in the injection. Then the hypercall function writes the respective parameters to a structure which, when the context switching occurs, the structure is read, and if all conditions are fulfilled, the bit-flip is performed [21].

Moreover, the last available tool is **Software fault injection in C source code**. This tool uses the source code of any program written in C language and applies modifications. The fault model used for the respective modifications is the Durães et al. fault model mentioned before since it is one of the most representative of software faults. First, the tool produces the abstract syntax tree of the input source code through a lexical and syntactic analysis. Later, it searches on the tree to identify potential nodes to FI, modifying the nodes with faults and converting the tree back to source code representation [21].

3.2 Fault Injection Acceleration

Although FI is a beneficial process for evaluating systems and fault-handling mechanisms, it is very time-consuming. As mentioned, a FI experiment evolves a location, the timing of injection, and fault type. The combination of these three variables makes a target system present a vast number of FI possible experiments that may take too long. Furthermore, it may be inaccurate and inconsistent since there might be locations that are never executed and other locations that are used very often.

In order to accelerate and achieve good accuracy in FI campaign, there are multiple types of research in FI acceleration field, both for hardware and software faults. These techniques always aim to accelerate the FI campaigns. However, acceleration has multiple meanings, such as reducing the time taken to complete a fault injection campaign or maximizing the number of failures (thus requiring less experiment runs to obtain a certain amount of failures).

The following section demonstrates some FI acceleration techniques for hardware and software faults that vary in the acceleration type.

3.2.1 FI Acceleration Techniques of Hardware Faults

Behrooz Sangchoolie et al. [1] presented two pre-injection techniques for improving controllability and efficiency in Instruction Set Architecture level FI (i.e., injection in CPU registers and memory locations reachable from software). Pre-injection techniques are executed before the realization of a FI campaign to analyze the target system and its specific characteristics.

The first technique is **Data Type Identification (DTI)** which is used to improve controllability which is, as mentioned, the ability to control the time and location of the injection. This technique identifies data-items (the register or memory content which can be a data variable, memory address or control information) where faults will be injected. This technique presents two identification methods: Instruction-Based and Location-Based DTI [1].

In **Instruction-Based DTI**, by knowing the machine instruction type used for accessing some location during the system execution, it is possible to identify some types of data-items. For example, the *lbz* instruction in assembly code loads a byte from memory, meaning that a register used in that instruction is an Address data-item (a value that represents the location of a part of data in memory). Whereas in **Location-Based DTI**, it is possible to know the data-item type that a register or memory segments hold since some locations always hold the same data-item type. For example, in an assembly code, a register can be categorized as a Text segment address data-item when used in an *mtlr* instruction since this instruction places a general purpose register into the link register (LR), which is always pointing to somewhere in the Text segment [1].

The other technique is **Fault Space Optimization** which is used to improve the efficiency of FI campaigns by removing bits of data-items that would always raise an exception in the system's execution, thus accelerating the campaign [1]. An example provided by B. Sangchoolieis et al. [1] is the Stack segment address data-items, where the bits 17 to 22 would always raise a hardware exception.

In another article, Behrooz Sangchoolie et al. compare the efficiency of inject-on-read and inject-on-write techniques in FI at ISA-level [22]. As the name says, inject-on-read means injecting a fault immediately before reading the respective data-item. This technique has two variants: **unweighted inject-on-read**, where all faults targeting the same location are considered equivalent. The problem with this technique is the possibility of unrealistic results since it is probable that some locations are more likely to be targeted by a fault than others because some locations contain more instructions, thus being more vulnerable to faults. Therefore, another used variant is **weighted inject-on-read**, where weight is attributed to each location according to the probability of the occurrence of a fault so that a fault that occurs less often is not as significant in the analysis as one that has a higher probability of occurring.

Inject-on-write means injecting a fault when the data-item is updated and written back into a register or a memory word [22]. In this technique, generate a block of instruction where it is known that any fault that targets one of that instructions will only produce some effect after the first write of the respective register. Therefore, we can avoid the injection of faults on those instructions set.

Another experimental acceleration technique is the FI with **failure models** [21]. The injection of failure models means that instead of injecting faults that can crash a process or corrupt a memory location, we directly inject these effects to study the system's behavior. For example, a process crash, since it is a common failure model in single bit-flips injection. P. Almeida et al. [21] performed an experience in order to compare FI with fault and failure models. This experimentation was conducted in the ucXception framework due to its ability to integrate new techniques. They used a cloud environment as setup where the target system was a cloud operating system called Openstack, and the workload was some typical operations of an Openstack administrator. The failure model was a crash of a random process. They concluded that failure models generate more failures than FI with fault models and that the generated failures may differ from those when fault models are used [21]. However, this experimentation had some drawbacks, like the failure model was too restricted since they only crashed random processes, they made only a few experiments, and the used setup was limited thus, the results may not be so exact.

Alfredo Benso et al. presented a **set of collapsing rules** that aims to reduce the fault list size and time by analyzing the assembly code and the behavior of a free fault system run. These rules aim to avoid the injection of faults where their behavior can be predicted. Therefore, the rules for the reduction of the fault list are based on three criteria: 1) the fault will be detected by at least one detection mechanism, 2) the fault is from a fault equivalence class, and the fault list already contains another one from that class and 3) the fault will not produce any effect on system's behavior and therefore can be removed [23]. With the reduction of the fault list size, the time needed for the experiment will be less since there are fewer faults to inject.

3.2.2 FI Acceleration Techniques of Software Faults

Erik van der Kouwe et al. propose a new approach called **Hybrid Software FI (HSFI)** [24] that uses the source-level context information for the FI, enabling and disabling the introduced faults without needing to rebuild the system. It injects all the faults at once but in disabled mode. They are activated later one-by-one by modifying the machine code. Also, it generates two versions of the code for each injected fault, a pristine version containing the original operations without the fault and a faulty version containing the mutated operations. Then, to identify the pristine and faulty versions, it generates a marker in order to the binary level identify each version. In short, the idea of HSFI is to have all the faults in the executable file and thus avoid the recompilation of the code multiple times (which is a time-consuming process) [24]. However, the representativeness of the system drops drastically since the system is run with a large number of faults embedded.

Another approach was presented by Robert Natella et al. [12], who studied the representativeness of more than 3.8 million faults and test cases in three real systems (two Data Base Management Systems and one Real-Time Operating System) with the help of the G-SWFIT [19] approach. The goal was to understand how many of them can be defined as residual faults, i.e., faults not revealed by test cases performed during development. The results show that for the DBMS, the percentage of representative faults, i.e., residual faults, varies from 14.57% and 23.13%, and in the case of an RTOS is 72.23%. Then they propose two artificial intelligence (AI) classification algorithms to inject only the most representative faults. The first one is based on **decision trees**, a supervised classifier trained by providing examples of components. The second one is **k-means clustering**, an unsupervised classifier that depends on observing the component less exposed to testing [12].

Stefan Winter et al. present a technique to reduce the time needed to perform a FI campaign, which consists of the parallelization of FI experiments called **Parallel Fault Injection (PAIN)**. Due to hardware technological developments that led to multi-core systems, the parallelization of experiments can be a method to increase its efficiency. However, this technique has a crucial point: relying on the assumption that the execution of multiple experiments in parallel does not affect the validity of results. In [25], a study is conducted to validate the accuracy of a FI campaign and the importance of the degree of parallelization to the efficiency and consistency of a campaign. They concluded that the PAIN technique significantly improves time efficiency. However, it degrades the representativeness of the results because of the degree of parallelism [25], thus being a crucial point of this approach.

Artificial Intelligence has been one of the most promising areas to FI optimization. Ali Sedaghatbad et al. propose a deep learning method for fault space exploration called **DELFASE** [11]. It consists of Generative Adversarial Networks for the identification of critical faults (faults that reveal system vulnerabilities leading to the violation of safety requirements). A GAN model is composed of a generator and a discriminator. The goal is to make the generator capable of generating dummy data that the discriminator cannot distinguish from the real one. This technique has two modes: an active, where a ranked batch-mode sampling is used to select faults for the GAN model training, and a passive, where the faults are selected randomly [11].

Table 3.1 presents the FI Acceleration techniques mentioned above and the articles where we can find more details.

Fault Type	Acceleration Technique
Hardware Faults	Instruction-Based Data Type Identification [1]
	Location-Based Data Type Identification [1]
	Fault Space Optimization [1]
	Unweighted Inject-on-Read [22]
	Weighted Inject-on-Read [22]
	Inject-on-write [22]
	Fault Injection with Failure Models [21]
	Fault List Collapsing Rules [23]
Software Faults	Hybrid Software Fault Injection [24]
	Fault Representativeness Selection [12]
	Parallelization Fault Injection [25]
	Fault Space Exploration: DELFASE [11]

Table 3.1: FI Acceleration Techniques

We can conclude that multiple techniques for FI acceleration exist, and each has its vantages and characteristics. However, it is an area that presents much potential where there are currently numerous types of research to optimize FI campaigns.

3.2.3 Failure Injection

Our approach is based on failure model injection. However, there are already some works regarding this injection type that use the injection of failures in multiple contexts to evaluate systems. However, there is no validation and comparison of how accurate the failure model can be compared with FI. Therefore our work aims to analyze the relations and differences between these two injection types.

Pallavi Joshi et al. proposed PreFail [26], a programmable failure-injection tool with the goal of avoiding the combinatorial explosion of experiments. This tool allows writing policies to define the set of possible failures. Their research defined pruning policies for distributed systems and evaluated the speed-ups with the respective policies in three cloud software systems. The policies are code-based rules with specific goals, such as filtering, clustering failures, or some probabilistic criteria. Their experimental evaluation compares their multiple policies technique with exhaustive testing.

Francisco Gortázar et al. introduced ElasTest [27], a platform for injecting failures intending to simplify the testing of distributed applications. This tool allows the users to test the application in every development phase of the lifecycle. With this open-source project, we can inject failures like packet loss, and node failure, among others. Their research demonstrates two different executions where a packet loss was simulated.

Big companies worldwide use failure injection to test their services in failure scenarios to ensure the system's availability. This concept is called Chaos Engineering [28], and it is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production. Netflix's first approach, Chaos Monkey, consists of testing its systems by randomly terminating a virtual machine hosting the production services. This technique was a success for Netflix, and therefore it was improved for testing other components and ensuring their reliability. Such success led other companies to create their testing programs, like Google and Amazon, which created GameDay [29] intending to ensure the resilience of their systems by injecting major failures in critical systems. In a retest, Amazon noticed that severe failures from two to three years ago now require less effort to solve than before GameDay was created [29]. Microsoft, inspired by Netflix Chaos Monkey, uses Azure Search Chaos for the controlled inject controlled disruptions and to simulate failures in a test environment in order to identify the weaknesses and defects. For Azure Search developers, this chaos engineering has been beneficial for developing reliable and fault tolerant cloud services [30].

As presented, the injection of failures has been used to validate the availability and resilience of systems. In our approach, we will inject failures and verify the potential of failure injection as a FI acceleration technique, i.e., in which situations can failure injection replace traditional FI.

Chapter 4

Approach

FI is a lengthy process that may take considerable time and even become infeasible since the injection of a fault in a system may have such a huge number of possible combinations (e.g., among fault location, timing, or types of faults) that the fault injection campaign would last for several years. Therefore, research on how to accelerate fault injection campaigns is essential. There are already multiple FI acceleration techniques, most of which are based around pruning useless fault locations that are never executed during execution or the parallelization of experiments. Although the goal is always to accelerate the FI experiments, it is possible to distinguish FI acceleration techniques according to their specific acceleration objective:

1. Reducing the time needed for FI while maintaining the representativeness of the results (e.g., the same failure modes and percentages);

2. Maximize the number of failures with the same number of FI experiments in a FI campaign.

The first acceleration type seeks to reduce the time spent in FI without losing the percentage of failures found before the acceleration since if the technique is capable of accelerating the FI but also reduces the failures found, then the FI campaign will not be reliable. Techniques like inject-on-read and inject-on-write [22], HSFI [24] or PAIN [25] are included in this acceleration type.

The other acceleration type aims to maximize the number of failures found with the same amount of FI experiments. This goal can be seen as an acceleration type because the more failures found in a fault injection campaign, the more effective it is for evaluating and analyzing the target system. This acceleration type is particularly useful for testing fault tolerance mechanisms, error detection mechanisms, or for producing failure data (e.g., to train failure prediction models). This category comprises techniques like DELFASE [11] or failure model Injection [21].

4.1 Failure Model Injection

Our research primarily focuses on the injection of failure models. The injection of failure models is an acceleration technique that has already been evaluated by P. Almeida et .al [21]. However, this research was brief due to time limitations.

The injection of failure models consists of, instead of injecting faults that might generate a failure in the target system, it is immediately injected the possible failures of a fault. Thus, skipping the FI and avoiding the injection of faults that are never activated and thus do not affect the target system. Furthermore, this technique is suitable for the evaluation of fault tolerance mechanisms because, with the immediate injection of the effects, we can realize and study the impact of failures and how well the mechanism is capable of tolerating them.

Although the FI acceleration technique is not dependent on a specific setup, the existence of a setup is helpful for the validation of the approach since it allows a comparison of the results obtained with our approach and without it. Therefore, we have decided to evaluate our FI acceleration approach using an existing setup that has been used in previous FI studies [19] and where we will measure the improvement associated to our approach. The setup is composed of two hypervisors containing multiple Virtual Machines (VMs) that migrate from one to another when a failure occurs through a technique called Romulus, which was designed to tolerate hypervisor failures. In this setup, FI is used for the validation of the fault tolerance mechanism that performs the VM's migration process [20]. The idea of the usage of a setup is for the comparison of FI outcomes with the results obtained with our approach in the same scenario.

4.2 Validation and Comparison to Fault Injection

This section defines all the aspects regarding the experiments for the validation and comparison of the failure model injection with FI.

4.2.1 Setup

The setup comprises two hypervisors, one **active** that holds VMs doing a specific workload and another **idle**, where VMs migrate when a failure occurs in the active one [20]. The hypervisor is a software system for the execution and management of VMs.

The hypervisors are hosted by a **microvisor**, which can be defined as a software layer that is vital to the migration process of VMs from one hypervisor to the other and also for the host of the two hypervisors in the same physical hardware machine [20].

For realistic experiments, the target system must be executing a workload. In this setup, the used workload emulates a Solr server, an open-source platform used for text-based searches, which will be exercised during the workload. Our VMs host a Solr server, and multiple clients can make requests to them. Our workload profile consists of one client making a request each second to the Solr servers in each experiment [20]. The workload is set to last 600 seconds when the experiments are hardware faults or failure models and 1200 seconds when it is software faults. This difference is because the software fault typically has a longer manifestation latency than hardware and failure models [20].

Figure 4.1 presents the described setup architecture [20]. In this scenario, the active hypervisor contains three VMs, when a failure occurs, it is expected that all of them migrate to the idle hypervisor. However, our setup will be composed of four VMs.

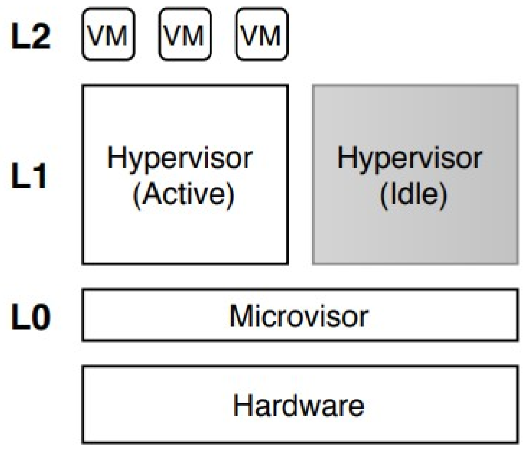


Figure 4.1: Virtualized system setup architecture [20]

For the detection of a hypervisor failure, it is necessary to have a detection mechanism capable of monitoring the active hypervisor so that the recovery process may start when a failure occurs. The selected mechanism consists on a continuous stream of ping requests to the hypervisor, and after some pings timed out, the trigger is activated. Thus, it is assumed that a hypervisor failure has occurred [20].

4.2.2 Recovery Mechanism

The recovery process consists on the migration of VMs held by the active hypervisor for the idle hypervisor, which will become the active one after the migration. This process is accomplished by Romulus, a tolerance mechanism used for hypervisors. Its goal is to migrate the VMs from the failed hypervisor to the idle hypervisor while avoiding the failure of the hypervisor from propagating to the VMs and corrupting their state.

For the migration, Romulus extracts the state of each VM running on the failed hypervisor and then resumes the respective VM's states in another hypervisor. This mechanism achieves a low downtime where the majority of time is spent on the state migration. Thus, the VMs can resume operation immediately after the migration and without a reboot requirement [20].

According to Frederico Cerveira et al. [20], Romulus is generic enough to be implemented on any hardware architecture (e.g., x86, Intel, AMD CPUs) hypervisor or any other considered software [20]. However, there are some specific basic requirements: A microvisor for failure detection and migration process management must support nested virtualization and virtual machine introspection (VMI) to allow the migration between hypervisors. It must be instantiated two hypervisors above the microvisor. The hardware where the setup is built must support hardware-assisted virtualization. Moreover, the VMs must be virtualized using hardware-assisted virtualization. Most of these requirements are usually present in real deployments. Thus, Romulus can easily be used for these systems [20].

4.2.3 Fault Injection

For the original evaluation of Romulus performance, FI of transient hardware faults in CPU registers and software faults in the hypervisor's code was used. These experiments lasted eight months. In this FI campaign it was injected more than 2000 hardware faults, which generated 774 failures, and more than 400 software, which resulted in 117 failures [20].

To inject hardware faults, the single bit-flip fault model was used, targeting the CPU registers of the hypervisor, namely, the registers RIP RSP, RBP, RAX, RBX, RCX, RDX, and R8 to R15. While for the software faults, the fault model proposed by Durães et al. [18] was used. Fault injection generates patch files that are subsequently applied to the source code of the active hypervisor. There is already an FI acceleration before the experiments are started an analysis of the source code of the hypervisor is performed to calculate which lines are never executed by the workload and then all the faults that do not affect those lines are drawn from the experiences [20].

4.2.4 Failure Model Injection Validation

Our main goal is to validate that our technique accelerates FI experiments. Therefore, we will compare our approach with the FI mechanism already implemented in the virtualized system.

To achieve that goal, we must define the targets and metrics that we use to evaluate and compare the performed experiments. Therefore, the possible targets where faults and failures will be injected are:

- Hypervisor
- Domain-0 VM
- Virtual Machines

Figure 4.2 shows a more detailed setup architecture and the possible injection points mentioned above.

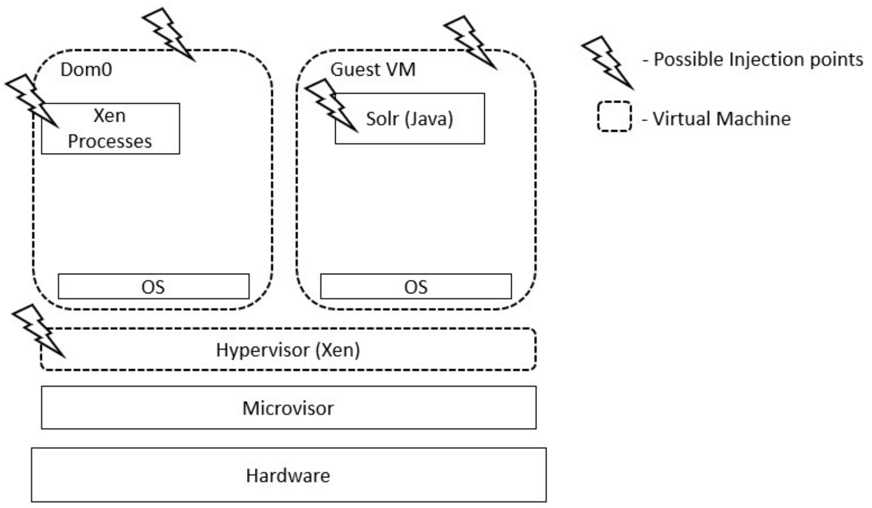


Figure 4.2: System detailed architecture [31]

The first target we considered for the experiments is the hypervisor as a whole, i.e., the injection of failures is executed in the hypervisor component without considering the specific components that compose it. This choice is based on the fact that the hypervisor is the critical point since its failure leads to the VMs not being able to function correctly. Moreover, the hypervisor is the single point of failure. Thus, all VMs are affected if it fails. Therefore, the final client (the person or component performing actions in VMs) notices that anomaly and cannot execute its tasks. Furthermore, the Romulus fault tolerance technique covers against hypervisor failures, and the original evaluation injected faults in this component [20], hence our choice.

Another possible target is the Domain-0 VM, this VM is responsible for managing the other virtual machines that run on the same hypervisor (Xen) and for providing the device drivers that will be used by the other VMs. Thus, it is a possible target for the failure model injection since it is essential for the system's proper functioning. We can inject failures in a specific part of the Domain-0 VM like Xen processes or in the VM as a whole system.

The other component where it is possible to inject failures is the VMs running the Solr server, where the client performs routine tasks. So the possible injection points are the Solr server and the VM as a whole system and its processes.

It is worth noting that the VMs where the client is using the service are our observation point since the goal of the fault tolerance mechanism is to keep the service's correctness and availability to the end user.

Figure 4.3 presents the experimental setup used for the experiments [20], where we can see two nodes, the **compute node** that represents the setup already detailed and the **orchestrator node** where the FI process is triggered by a FI manager, and later, Experiments Manager is responsible for saving the results of the experiments. There is also the Solr Client, our observation point, which performs

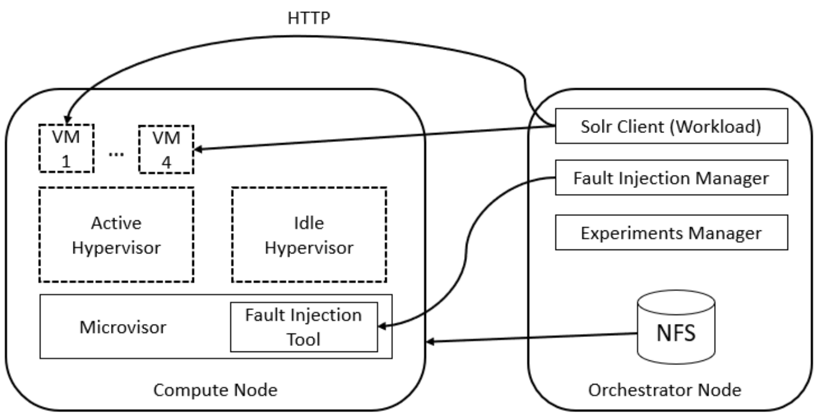


Figure 4.3: Experimental Setup [20]

the routine workload tasks by making HTTP requests to the multiple Vms. The NFS is a network file system in the orchestrator node and is used to take disk images to the VMs.

4.2.5 Failure Model

For the injection experiments, a failure model must be defined, which means the multiple failures type possible to be injected into the target system. Our failure model consists of three failures type:

1. Crash
2. Hang
3. Timeouts or failed hypercalls

They are ordered by their importance to the experiments. We considered the crash the main and the first that we evaluated. This failure can be performed in the hypervisor or VMs. A **crash** can be made in various forms, like a process kill or a kernel panic. The reason for being the principal failure is that it is the most common way of failure of a cloud system [31] [32] [33].

The second failure type is the **hang**, which is not as common as the crash but can also occur in the system. This condition can occur due to infinite loops, making the system unresponsive and failing.

The last failure type is the **timeouts or failed hypercalls**. This failure occurs when a request made by a VM to the hypervisor is not completed within a specific period, or the request is never completed, and therefore it is unsuccessful. Initially, we had considered this type of failure in our tool. However, we did not implement the respective approach due to time constraints, leaving it for future work.

For the crash failure injection, we forced a kernel panic in the respective machine that can be a hypervisor or a random L2 VMs [34]. This is a typically used approach for testing approaches.

While for the hang failure, we have created a loadable kernel module, which will stop all CPUs in the machine, thus making the system becomes unresponsive since the CPUs are occupied in an infinite loop and cannot process any further instructions or handle other tasks [35]. As the crash, the hang failure can be injected in the hypervisor or an L2 VMs.

As mentioned before, our failure model injection tool was also integrated into the ucXception tool, making it available for other researchers and testers.

4.2.6 Metrics

For a rigorous comparison, some metrics must be specified, aiming for the validation and comparison of the impact and the obtained results. We define four possible metrics:

- Manifestation Latency
- Percentage of recovered VMs
- Downtime
- Failure percentage with the same number of injections

The first metric is the **manifestation latency**. This latency refers to the time between the injection and the moment that the effect is perceptible in the target system. Since, in our experiments, the observation point is the Solr Client (the point where we detect the failure), then, our manifestation latency is calculated from the client's viewpoint. This time may be significant in the traditional FI experiments since we inject faults that may lead to effects and may never raise an effect. However, the injection of failure models bypasses this latency since we inject the effects of a fault immediately. Figure 4.4 schematizes the manifestation latency of a FI.

Another metric that we use is the **percentage of recovered VMs**. When a failure occurs in the active hypervisor, the Romulus mechanism performs the migration process. However, the process may not always successfully recover all VMs that the failed hypervisor contained. Therefore, we aim to keep this value in our failure model approach the same or similar to the same value at FI so that there is no loss of representativeness in our approach.

The third metric to be used is the **downtime**, which is the time that the system is down and unable to perform routine tasks in the VMs. This time value can be used as a metric since it must be equivalent to the time value obtained in FI. If the values are similar, then our technique provides good representativeness. However, if it is too different, that means that our approach is not reliable, and the results may be inaccurate.

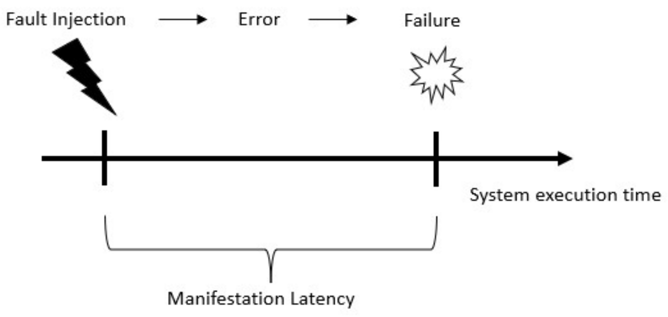


Figure 4.4: Manifestation Latency Scheme

Last, we can analyze the **failure percentage with the same number of injections**. We aim to maximize this value since the objective is to discover failures in the target system. If we find more failures in the same number of experiments with our approach compared with FI, then we are improving the experiments.

These four characteristics are the possible metrics for the comparison and, thus, validation of our approach

4.2.7 Fault and Failure Model Injection Flow

Each FI experiment is a complex process that involves multiple components. Hence, it is essential to present the flow of a typical experiment.

First, the user initiates a campaign, where he should indicate the injection type (hardware, software, crash, or hang), the number of VMs, and the number of experiments. In case of a failure model injection, the user also has to indicate the target, which can be the hypervisor (L1) or VMs (L2)

The campaign is initiated in our orchestrator node machine, responsible for managing FI experiments. Next, if it is hardware FI, the fault is generated, which means it generates an injection time sleep, and the target register and bit are chosen for the injection. The choice of register and bit is random within the defined fault model. While if it is a software fault, the patch file is chosen.

Then, the launch process of hypervisors is started, meaning, first, it is killed the hypervisors that might exist from the past experiments (in case the experiment ended incorrectly). The active hypervisor snapshots are mounted via a network file system where the snapshots are located. Finally, both hypervisors (active and idle) are created, and it is called a function that tries to make an ssh connection for both hypervisors to ensure that they have started correctly.

If we are injecting software faults, then the next step is to patch the active hypervisor XEN with the fault. This stage involves the compilation, build, and installation of the XEN hypervisor, which is a more time-consuming process, although necessary. If it is successful, the offsets are extracted, which can help further analyze and manipulate the fault injection process. Finally, the hypervisors are relaunched with the patched fault in the active one, and the experiment may continue.

When hypervisors are ready, the domain ID of both hypervisors is extracted. This ID will be needed for the injection of the fault or failure and the launch of monitors. Next, it starts the spawn of VMs in this process, like in the launch of hypervisors, it is necessary to mount the snapshots, and then the VMs are created.

With the hypervisors and VMs ready, the Solr service may start and save files created. The save files are where it is the necessary information for the migration process. This step is followed by the launch of the probes to detect failures in the VMs, and after a warmup phase, the fault or failure is injected. The workload is left to execute until the end. The recovery process is triggered if the monitoring probes detect that at least 1 of the VMs has failed.

After the ending of the workload process, the VMs state and correctness are verified through some tests. All the ping processes are killed, and if it is hardware FI, then the extraction of DMESG is performed. This log has the values of the registers after the injection. While if it is software FI, it extracts information about the fault activation.

Finally, the results are extracted and stored, and the physical machine is rebooted to avoid carrying over anything that may interfere with the following experiment run.

It is possible to analyze the diagrams of the different injection types flow in Appendix A.

Chapter 5

Results

In this chapter, we verify the capabilities of failure model injection in comparison with the traditional FI. Thus, we will start by characterizing the datasets of the results retrieved from the experiments explained in Chapter 4. Next, we analyze the four defined metrics, starting with the failure percentage, then manifestation latency, recovered VMs, and lastly, the downtime.

5.1 Dataset Characterization

From our experiments runs of fault and failure model injection, we obtained different datasets. These experiments took about three months to be completed, and each dataset can be seen as a cluster of experiments that use the same fault or failure model. We define six datasets: HW, SW, CRASH_L1, CRASH_L2, HANG_L1, and HANG_L2. This datasets are detailed in Table 5.1.

The HW dataset comprises experiments where hardware faults were injected into the active hypervisor according to the defined fault model. There are 300 experiments in this dataset.

The SW dataset is composed of 246 software FI experiments where the code of the active hypervisor was modified with faulty patches.

CRASH_L1 and CRASH_L2 datasets contain 297 and 300 experiments, respectively, where it was injected crashes. In CRASH_L1, the failure was injected in the active hypervisor like in HW and SW, while in CRASH_L2, the failure was injected in one of the possible L2 VMs.

The hang failure is represented in the datasets HANG_L1 and HANG_L2. Their differences are in the target component as in the crash datasets. HANG_L1 comprises 297 experiments, while HANG_L2 contains 299 experiments.

In order to understand the registers where hardware faults were injected, we show in Figure 5.1 the distribution of the different possible target registers. As expected, we see a relatively uniform distribution resulting from the random choice of target register.

Dataset	Experiments
HW	300
SW	246
CRASH_L1	297
CRASH_L2	300
HANG_L1	297
HANG_L2	299

Table 5.1: Datasets Characterization

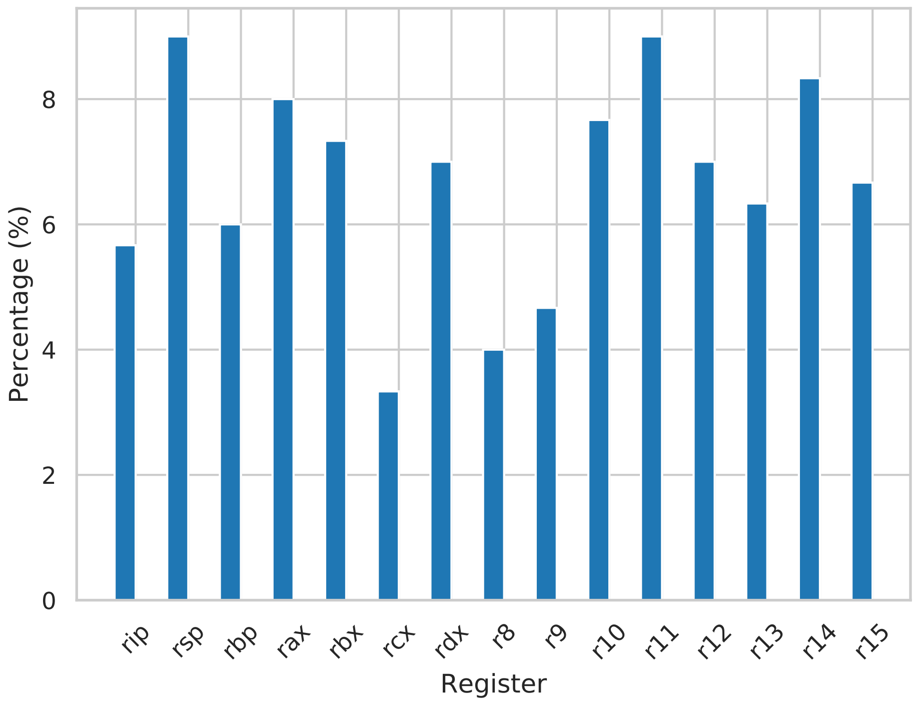


Figure 5.1: Register distribution in the HW dataset

A similar analysis was made for the distribution of the operators injected in each software fault. We can see that the patches MFC, MIA, and MIFS were the most injected ones. The distribution was not uniform, although it was expected since some patches are most likely to occur due to the pre-conditions necessary for the patch activation.

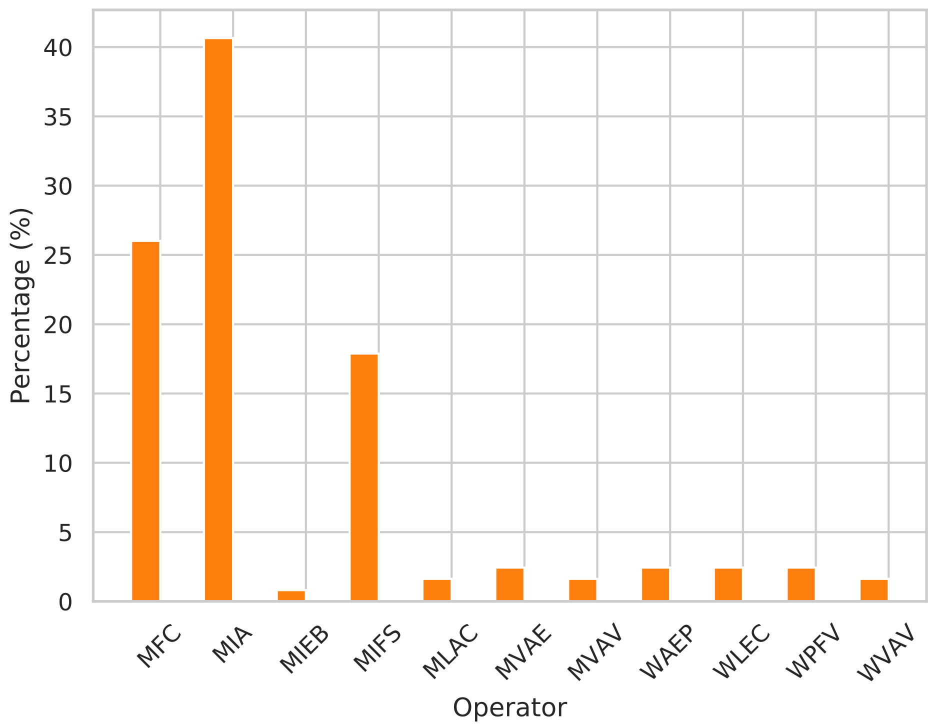


Figure 5.2: Operator Distribution

5.2 Failure Percentage

The first metric we are going to analyze is the failure percentage. This value represents the percentage of experiments that generated a failure in the system. Thus, a higher percentage of failures means that fewer experiments are needed to obtain results of the occurrence of failures. Table 5.2 shows all datasets, their number of failures, and the respective failure percentage.

As we can see, HW presents a failure percentage of 29%, while SW has 88.62%. Finally, the failure model injection datasets have 100% of failure injection. This percentage is expected since we directly inject the failure, i.e., the possible consequence of a fault, for example, the crash.

HW dataset has a lower percentage since the register and bit of injection are randomly chosen from within the defined fault model, and there are register and bits less significant for the injection. Unlike the SW dataset that presents higher values of failure percentage, this can be explained by the fact, as mentioned in Section 4.2.3, the software FI mechanism is already optimized, e.g., we do not inject faults in lines of code that are not exercised by the workload.

Dataset	Num. Failures	Failure Percentage (%)
HW	87	29
SW	218	88.62
CRASH_L1	297	100
CRASH_L2	300	100
HANG_L1	297	100
HANG_L2	299	100

Table 5.2: Failure Statistics per Dataset

From these results, we conclude that experiments where failures were injected had perfect effectiveness, representing an improvement close to 3x compared to hardware FI (29% to 100%). While compared with software, it is 11% over the already optimized injection of software faults.

5.3 Manifestation Latency Analysis

The second metric we analyzed was the manifestation latency, i.e., the time between the actual moment of injection and the first perceptible sign of failure. For this analysis we filter out experiment runs where no failure was detected.

We started by calculating the manifestation latency using the failure detection times, which are calculated through pings (after 40 failed pings requests, we considered that there was a failure), which consists of performing multiple pings to L2 VMs. Figure 5.3 shows the distribution of the time of the manifestation for each dataset.

This figure shows that the SW dataset has the most significant difference among all datasets. SW's average manifestation latency time is 176.98s, whereas the median is 99.23s. This difference means the distribution is asymmetric, meaning there is a high deviation between the results. Therefore, the software experiments regarding the manifestation latency are less consistent than the others. However, software failures tend to have a longer manifestation time than hardware failures and contain more outliers than hardware failures with very long time values [36].

CRASH_L2 and HANG_L2 present the shortest times, where CRASH_L2 has an average of 53.72s and a median of 53.61s, while HANG_L2 has 53.12s and 53.70s, respectively. These times indicate that when a failure is injected in L2 VMs, the time between the injection and the failure is shorter. This behaviour is expected since when we inject a failure in the L1 hypervisor, it has to suffer propagation to reach an L2 VM. In contrast, if we inject directly in an L2 VM, propagation is unnecessary.

For a better analysis of the other datasets, we generated Figure 5.4, where the rest of the manifestation latency times can be seen in more detail. The timing measurements from the three datasets (HW, CRASH_L1, HANG_L1) display striking

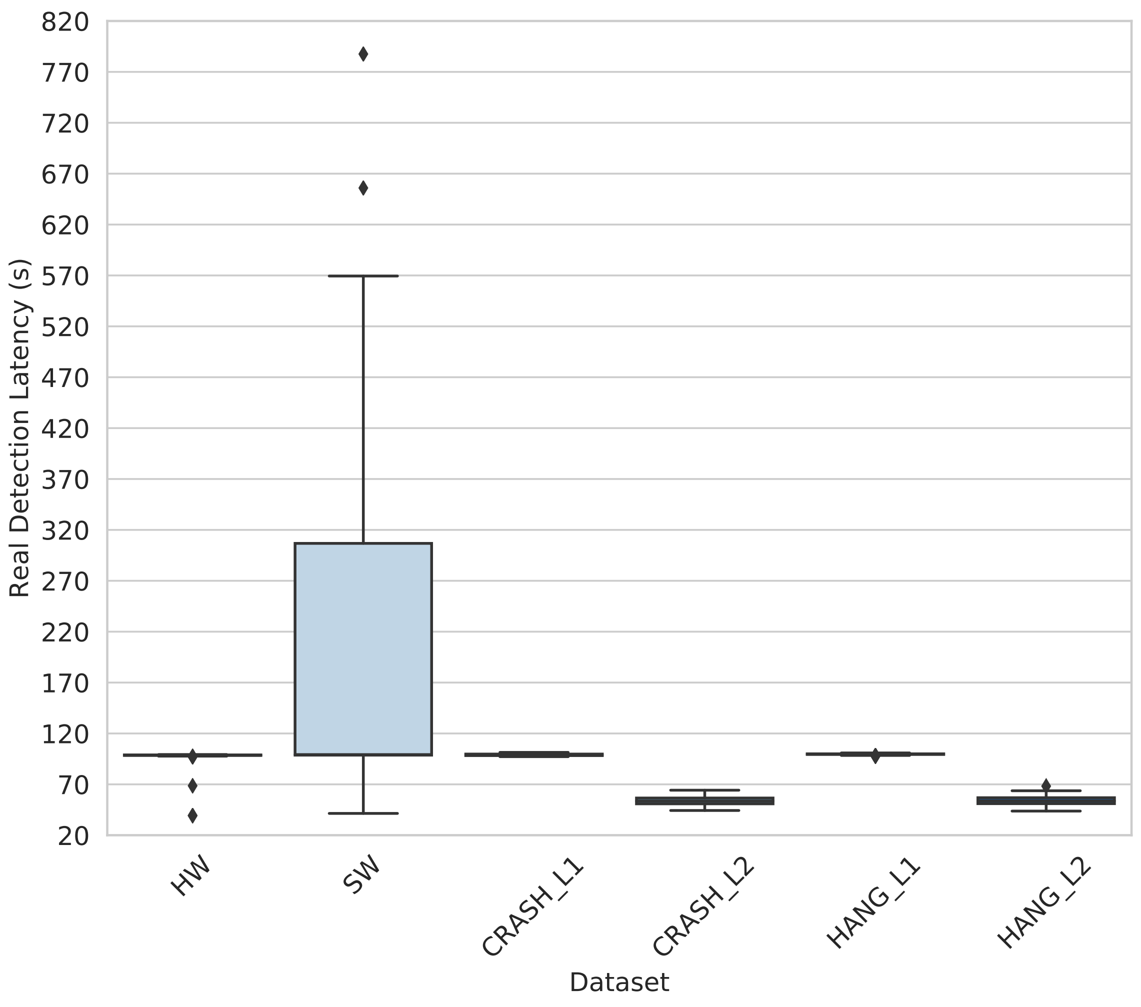


Figure 5.3: Manifestation Latency - Ping

similarities. The HW dataset shows an average time of 96.87s and a median time of 98.60s, while the CRASH_L1 dataset exhibits an average time of 99.06s and a median time of 98.94s. Similarly, the HANG_L1 dataset indicates an average time of 99.60s, closely aligned with its median time of 99.67s.

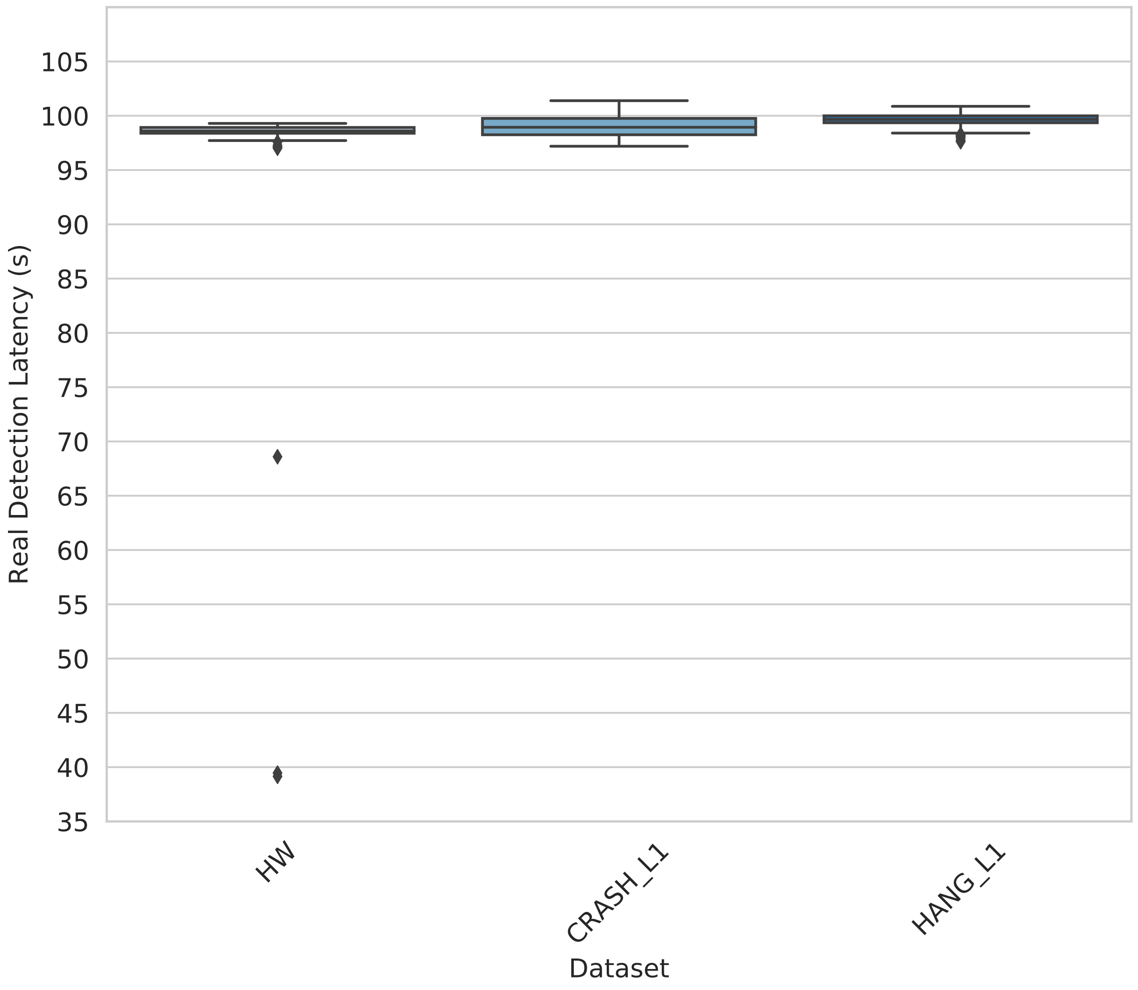


Figure 5.4: Manifestation Latency - Ping (subset of datasets)

However, as in the SW dataset, we can detect some outliers in the HW. This is to be expected since fault injection can be done according to various parameters (e.g., register and bit, software fault operator and code location), and thus their behaviour can vary. In failure model injection, on the other hand, the behaviour should be almost identical since the same type of failure is injected into the same component, resulting in minimal differences. However, it is necessary to consider the parameter of injection timing, as it can potentially impact the outcome.

The consistency of manifestation latency is a vital factor influencing the duration of an experiment run. If the manifestation latency is inconsistent, the person designing the campaign must consider the highest possible value when defining the amount of time the workload should run. Otherwise, if the workload is too short, some faults that would cause a late failure are cut abruptly and incorrectly labeled as having had no effect.

We expected the manifestation latency of failure injection to be very low (due to skipping the fault to error and then to failure chain), however, the earliest results did not confirm this assumption. Thus, we decided to calculate the manifestation latency from a different viewpoint, in this case, the workload clients themselves. In other words, we measured the latency between the injection moment and the failed first request. The results are shown in Figure 5.5.

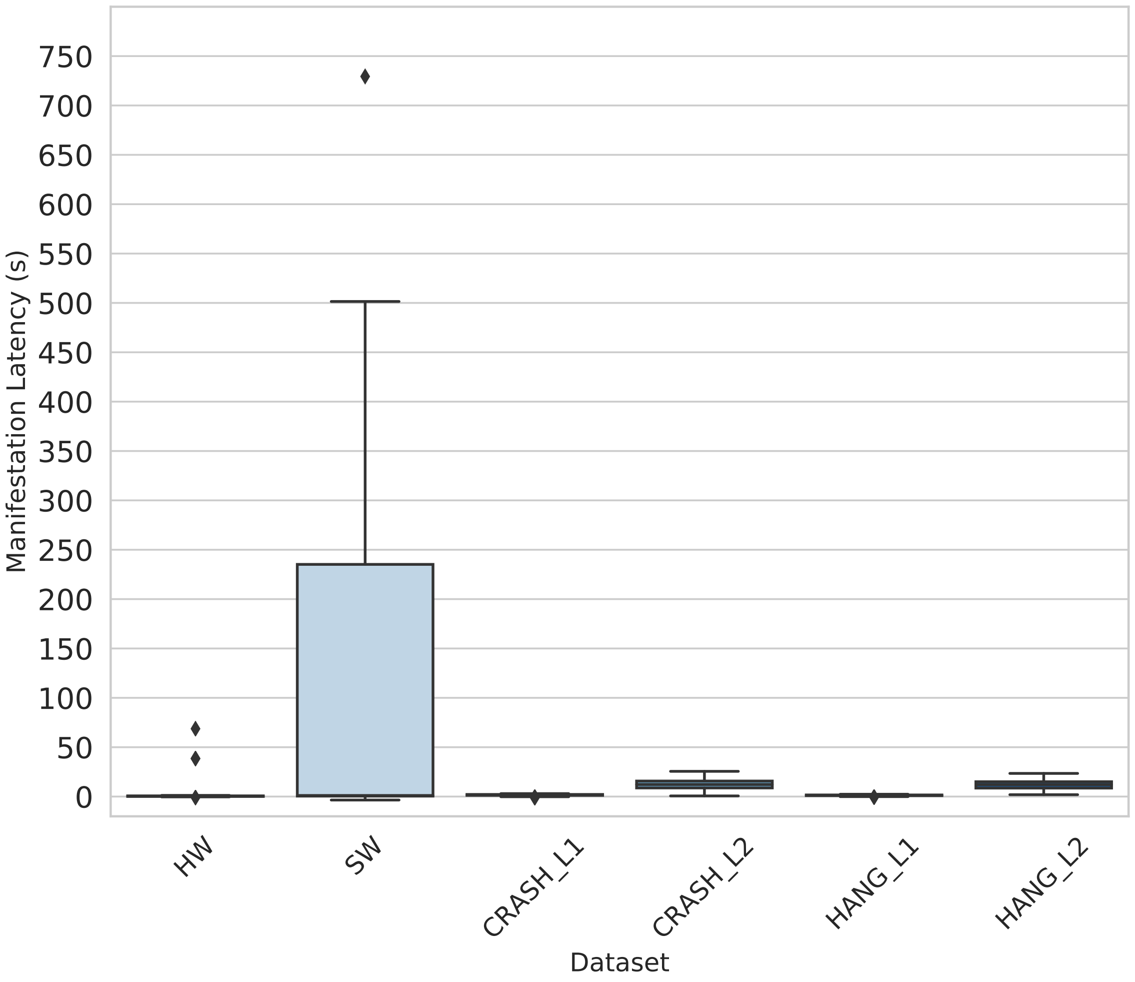


Figure 5.5: Manifestation Latency - Solr (Workload)

These results are most expected when we look at CRASH_L1 and HANG_L1, where the times are close to 0s. This difference between these results and the manifestation latency calculated through the failure detection mechanism can possibly be explained by the effect of a network timeout mechanism that affected the obtained latencies when looking only at the ping mechanism. As future work we aim to confirm this suspicion.

The SW dataset still has the longest manifestation latency time, as expected, with an average of 85.41s and a median of 1.10s, meaning that most experiments took about 1 second. Hence, the median is low, but some took a long time to generate a failure, so the average manifestation time is very long.

The CRASH_L2 and HANG_L2 datasets have longer manifestation latencies. On average, the crash of L2 VMs occurs after 12.40s with a median of 12.23s, whereas the hang has an average manifestation time of 11.15s and a median of 12.06s.

The manifestation of hardware failures in the HW dataset is similar to the times measured in CRASH_L1 and HANG_L1 datasets, as we can verify in Figure 5.6. The average manifestation time for hardware failures is 2.04s, with a median of 0.42s. Specifically, for CRASH_L1, the times are 1.63s and 1.71s, while for HANG_L1, they are 1.23s and a median of 1.29s.

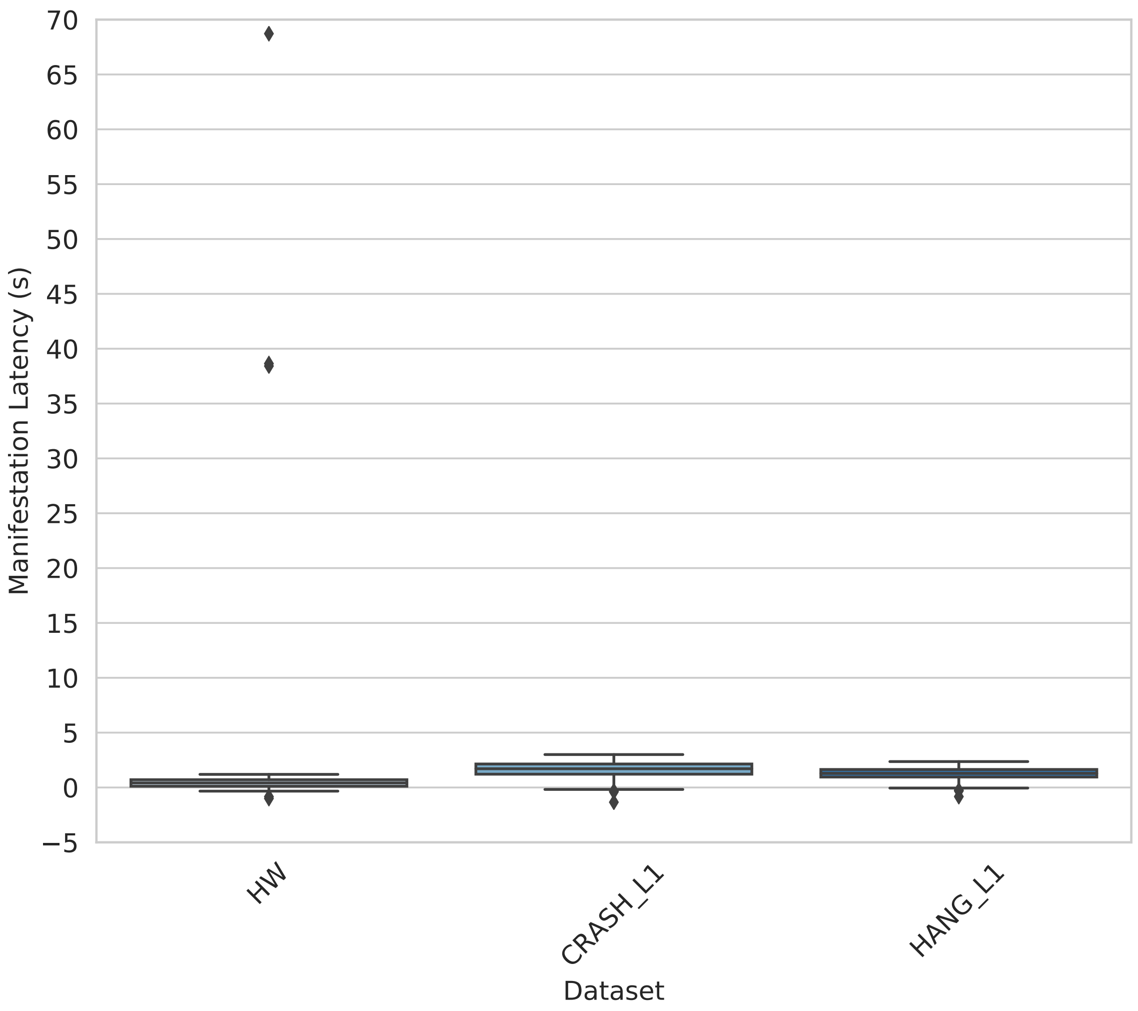


Figure 5.6: Manifestation Latency - Solr (Workload) (subset of datasets)

It is also possible to verify that there are some negative times in each dataset. Small measurement errors can explain these outliers since our environment is a distributed system, which is more susceptible to clock synchronization problems across different machines. The timestamps used for calculating this manifestation time are retrieved from the Orchestrator node and from the hypervisor, and both can present slight variations between them.

From these results, we can state that the injection of crashes and hangs in the hypervisor (L1) is a valid and faster alternative to injecting hardware faults since their since the obtained manifestation latencies are similar. However, the results from injection in L2 VMs have a significant difference, and thus we cannot conclude the same when the target is the L2 VMs. The same is valid for software fault injection, as its manifestation latency differs significantly from all the other datasets in the comparison.

Also, the ping mechanism used is not a reliable way of analyzing the actual manifestation latency since the ping mechanism can present delays in the measurement of manifestation time.

5.4 Percentage of recovered VMs Analysis

The recovery percentage of L2 VMs is an important metric to analyze since when the goal is to inject failure models to evaluate fault tolerance mechanisms, such as Romulus, we must guarantee that the result of the injections will be identical to the one produced by the fault models when there is a failure. Therefore we study the recovery percentage of L2 VMs by Romulus when we inject fault and failure models. We can analyze the recovery performance of Romulus from two points of view:

- Solr service
- Operating System (OS)

If we look at it from the Solr point of view, we see if the Solr service running on each L2 VM before the migration was recovered after the process was completed. From the operating system point-of-view, we check if the operative system kept working after the migration even if the Solr service did not. We make this distinction because there are some situations where Romulus can recover the operating systems but not the Solr service.

Regarding the Solr point-of-view, it is possible to observe the bar chart of the distribution of recovered VMs in Figure 5.7. The recovery of two VMs is the most common scenario, where CRASH_L2 and HANG_L2 have the highest probabilities with 42.33% and 47.16% respectively followed by HW with 37.93% and CRASH_L1 with 34.34%. SW and HANG_L1 have the lowest values, 27.98%, and 23.75%.

SW is the dataset with the highest probability of not recovering any VM with 21.10%. This value is significant when compared with the other datasets since HW and CRASH_L1 are the next ones with higher probability are 8.05% and 7.74%.

Injecting a failure into an L2 VM makes it impossible to recover in the migration process, as can be seen from the bar chart where CRASH_L2 and HANG_L2 have no probability of full recovery of the VMs. This observation is expected since the fault tolerance mechanism was not designed to recover from failures in L2 VMs, only from the faults and failures from the hypervisor.

From the OS point of view, Figure 5.8 shows the probabilities of L2 VMs operative system recovery. From the bar chart, we can see, as expected, the failure of L2 VM does not affect the other VMs OS performance. This is verified by the probabilities of recovery in CRASH_L2 and HANG_L2, which are 69.33% and 71.57%,

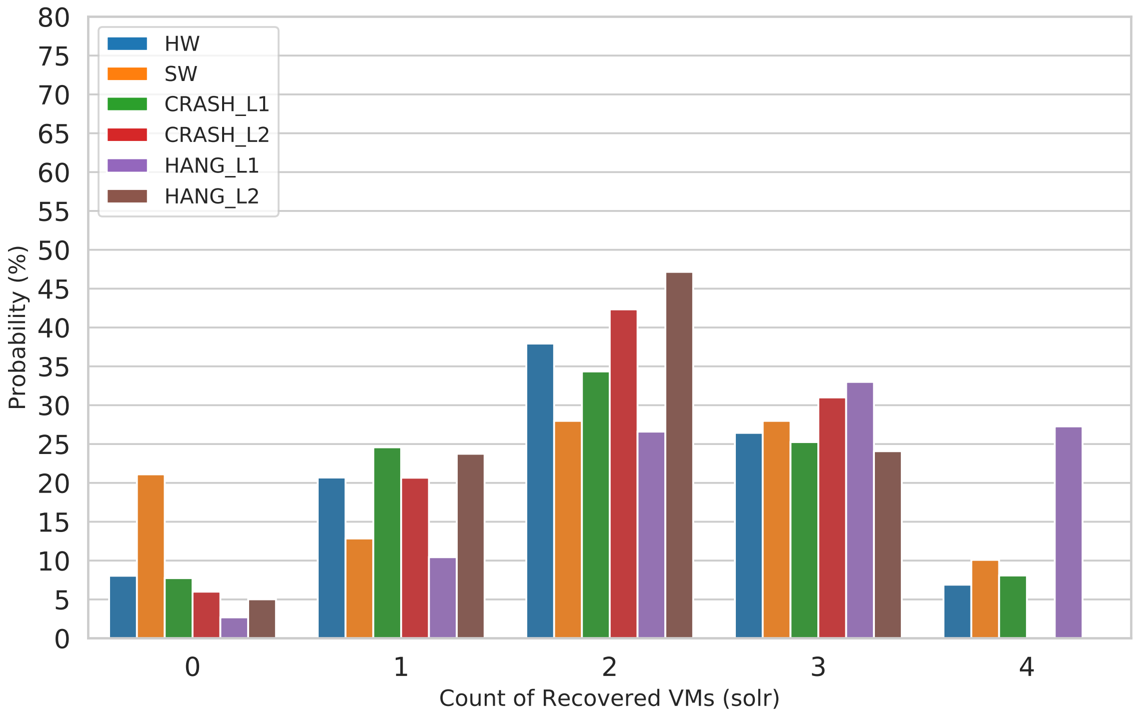


Figure 5.7: Recovered VMs (Solr)

respectively. The injection of a hang failure in L1 has a minor effect on the recovery of all VMs since the probability of a full recovery is 61.95%. This value is very high when compared with SW, which is the second injection type where the total recovery is more common, with 38.99%. Thus, it is worth noting that the hang failures present a higher percentage of recovered VMs than crashes.

As in the Solr service point-of-view, we can observe that HW and CRASH_L1 have the most similar probabilities of recovering VMs.

To better comprehend the results, we create a cumulative histogram of the probabilities of VM recovery, which can be seen in Figure 5.9 and Figure 5.10. From these histograms, we conclude that CRASH_L2, HANG_L1, and HANG_L2 are where the recovery is most successful. Interestingly, SW presents the lowest recovery capacity for one or two VMs, but for three or more, it has reasonable probability compared with HW and CRASH_L1.

These two histograms further highlight the similarity of results between the HW and CRASH_L1 datasets. As evidenced earlier, injecting crashes into the hypervisor (L1) can be a valid alternative instead of injecting hardware faults.

In summary, from these recovery results, we infer that HW and CRASH_L1 have similar probabilities of VM recovery. This conclusion reinforces the idea that using crashes as a failure model is a valid alternative to injecting hardware faults when the target is the hypervisor. Thus, if the goal is to study the efficiency and success of recovery from a fault tolerance mechanism, we can consider the crash model injection.

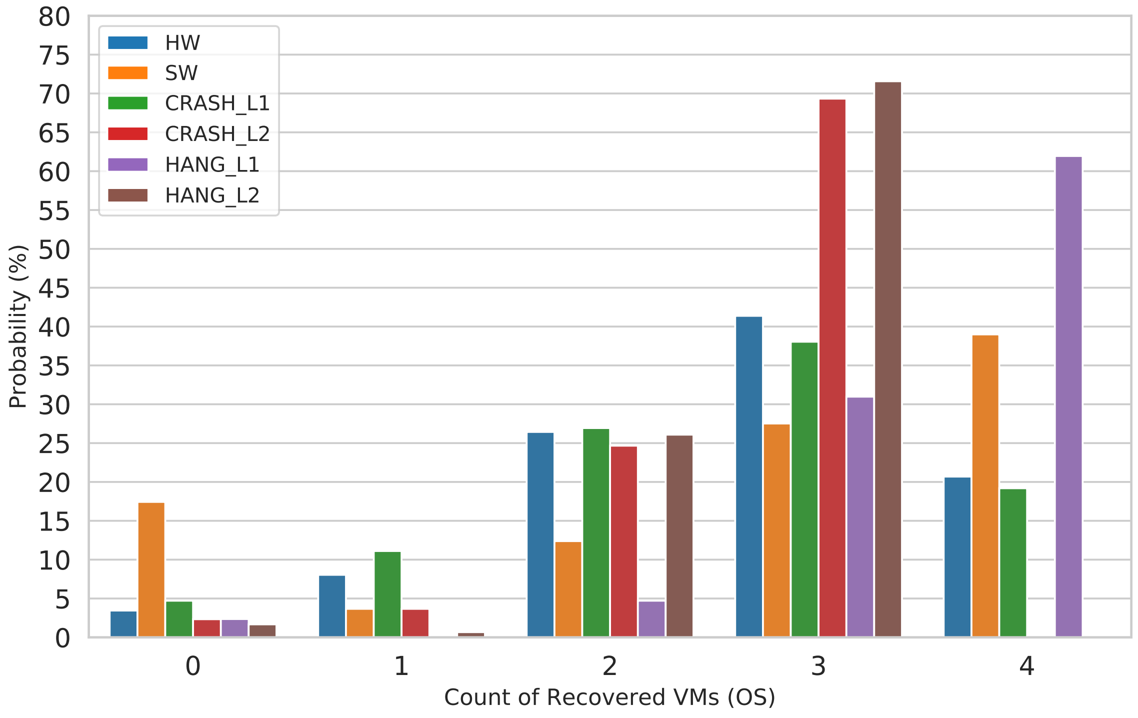


Figure 5.8: Recovered VMs (OS)

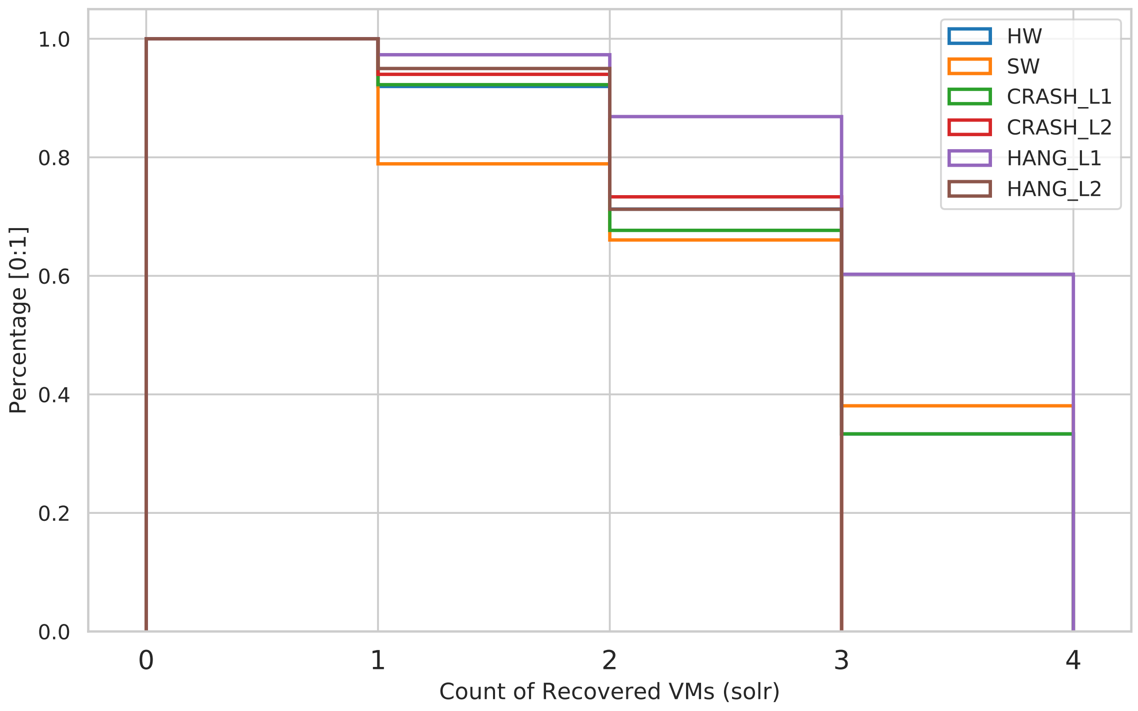


Figure 5.9: Cumulative Histogram Recovered VMs (solr)

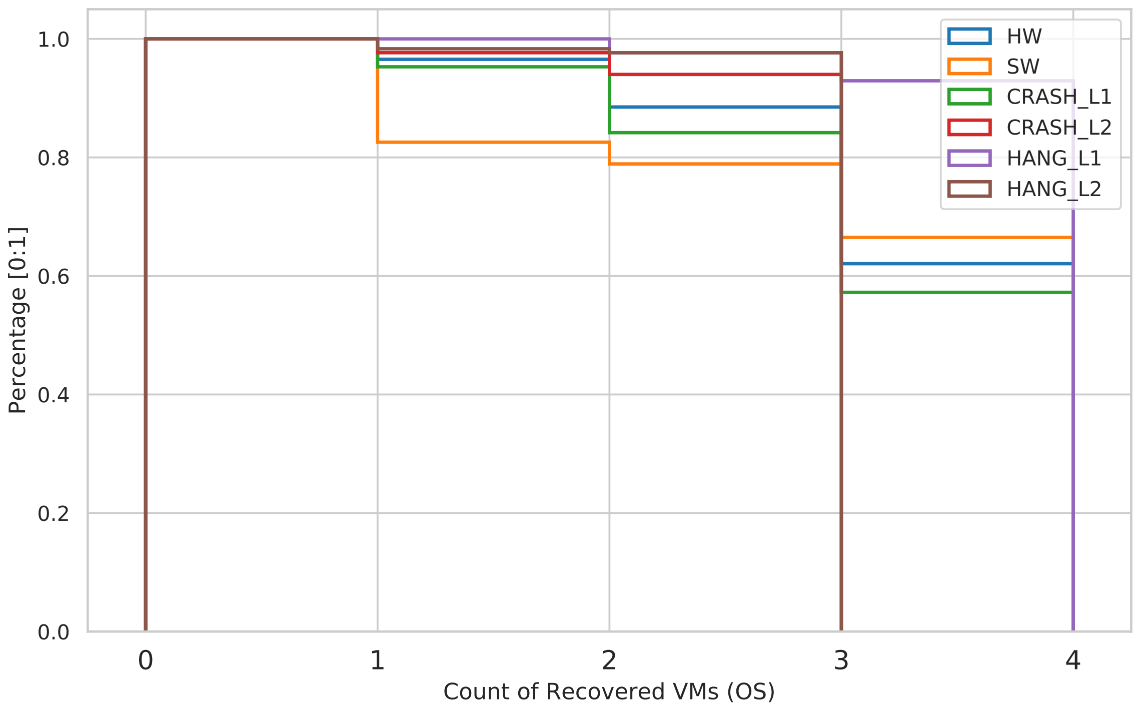


Figure 5.10: Cumulative Histogram Recovered VMs (OS)

5.5 Downtime Analysis

The last metric that we study is the downtime in the Romulus recovery process. The downtime is the time taken from migrating from the active hypervisor to the idle one. We study whether different fault and failure models impact VM downtime.

Figure 5.11 presents a box plot with each dataset's L2 VMs downtimes. The median values of HW and SW are 253.25s and 262.68s, while the CRASH_L1 and CRASH_L2 are 250.75s and 268.05s, respectively. Finally, HANG_L1 and HANG_L2 are 253.05s and 270.51s.

These results suggest that the downtime does not vary when failure model injection is used. Therefore, if the goal is to study the downtime of the VMs, both the failure model and fault injection can be considered since their results are identical. Thus, the type of injection does not significantly influence downtime.

5.6 Limitations

In every experimental work, there are always some limitations and improvements that could be made. The main limitation was the fact that we only performed the validation in one setup and one workload. Furthermore, this is a particular setup application: a node of a cloud computing deployment that hosts VMs that support a read-heavy Solr based workload.

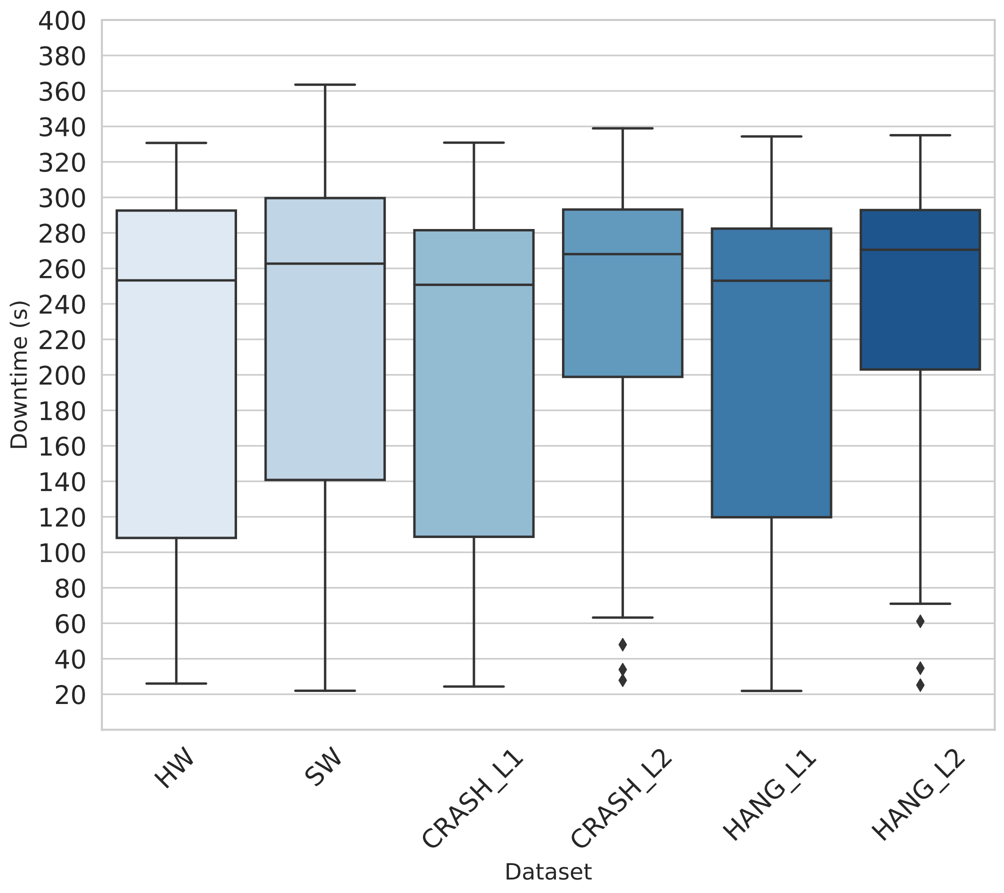


Figure 5.11: Recovery Downtime

Another limitation is the defined failure models. We had considered the two most commonly failures in virtualized systems (crash and hangs) however, there are other failure that could be considered such as disk error or memory. The failure model injection could also target more injection points such as critical processes running on hypervisor or L2 VMs or the Domain-0 VM.

5.7 Summary

We performed a study of the four different metrics by comparing six fault and failure model injection datasets and obtained interesting observations, which we summarize in this section. The first observation is that the crash failure in the L1 hypervisor produces similar behaviors to those resulting from the injection of hardware faults. This dataset's similarity is also notable in the recovery percentage VMs. At the same time, the downtime is similar for every injection type. Thus, we can conclude that the injection of a crash in L1 is a valid alternative to the injection of hardware faults since it produces similar results in all analyzed metrics: manifestation latency, percentage of recovery, and downtime. Hence, it can produce similar results at a much quicker pace (about 3x quicker). For the study of manifestation latency, the injection hangs in the hypervisor is also a valid alternative since it is similar to the times from hardware FI. Therefore when the goal is to evaluate a fault tolerance mechanism through hardware FI, we can replace this injection with the injection of crashes.

However, certain injection points, such as L2, and specific failure models, like hang, exhibited different behavior than traditional FI techniques. Therefore, they do not represent a valid alternative to injecting faults such as the L1 crash. Furthermore, none of the failure models injection could reproduce the results obtained from the software FI.

Injecting failures to evaluate the dependability of a system or design failure prediction mechanism is not a valid technique since dependability encompasses various concepts beyond fault tolerance, such as availability and reliability, and the injection of failures cannot evaluate these properties. And for the prediction of failures, failure injection is not valid since it does not provide patterns of failures like FI can reveal.

Chapter 6

Planning and Methodology

This chapter presents the planning work of this dissertation and the detailed work stages of each semester with the respective Gantt charts. It also presents the main risks associated with the planned second semester work.

6.1 First Semester

The first semester lasted 19 work weeks and contemplated a more theoretical side of the research. The Gantt chart (Figure 6.1) illustrates the first semester work that was divided into four stages:

1. **State-of-the-art research**: We started with readings about dependability, then the research about fault injection, and finally, fault injection acceleration.
2. **Approaches Planning**: In this work stage, we considered multiple acceleration techniques for implementation
3. **ucXception Study**: Since one of the goals of this dissertation is to integrate our approach to ucXception, it was required to make an introductory study about the framework.
4. **Intermediate Report Writing**: This was the last stage of the first semester, where all the knowledge gained during the semester was written.

During the semester, we had weekly meetings where articles were presented and discussed. Also, if there was any doubt, it could be discussed via e-mail on the other days of the week. At first, the supervisor, Frederico Cerveira, suggested reading articles that fit into the dissertation's scope and would be helpful in its development. Later in the semester, I started looking for new ones independently. All articles were noted in Excel with the name, author, and subject in order to simplify the process of writing the dissertation. All the planned work for the first semester was achieved.

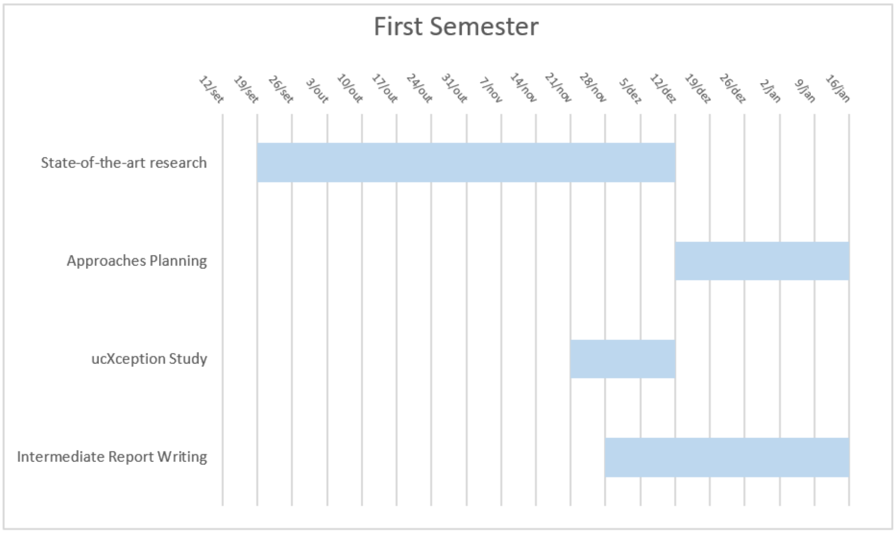


Figure 6.1: First Semester Work

6.2 Second Semester

The second semester work lasted 24 weeks and was spent preparing the setup, developing our failure model injection approach, and validating the results. The work stages of the second semester were as follows:

1. **Setup Preparation**: This was the first stage of the second semester, and it refers to preparing the setup for the experiments that will be used to validate our approach. This setup is based on a setup that has been developed and used in previous works. Hence, we had to check the hardware setup, then we checked and adapted the code of FI since we should strive to replicate it as closely as possible.

2. **Fault Injection Experiments**: After the preparation of the setup we performed some FI experiments using traditional FI (i.e., using fault models) in order to collect data to validate that the setup was correctly replicated and to compare with our approach.

3. **Failure Model Injection Tool Development**: While the experiments mentioned above are being performed, we started the development of our failure model injection technique for accelerating fault injection.

4. **Failure Model Injection Experiments**: This stage comprises the campaigns of failure model injection. For this phase, it was necessary to have our technique fully developed.

5. **Paper Writing** While the failure injection experiments were running, we started writing the paper based on our dissertation work, where we validated the failure model injection compared to traditional FI.

6. **ucXception Integration**: We integrated our approach into the ucXception framework for future use.

7. **Dissertation Writing**: This was the dissertation's final step, which describes all the work and conclusions of the work developed during the year.

In the second semester, we kept the weekly meetings in order to discuss the problems that arose during the various work stages, such as error codes or the Romulus. Furthermore, we followed an experimental method, i.e., we collected fault and failure injection results to validate our approach and its reliability through defined metrics.

The following Gantt chart (Figure 6.2) presents the work stages we had foreseen in the first semester.

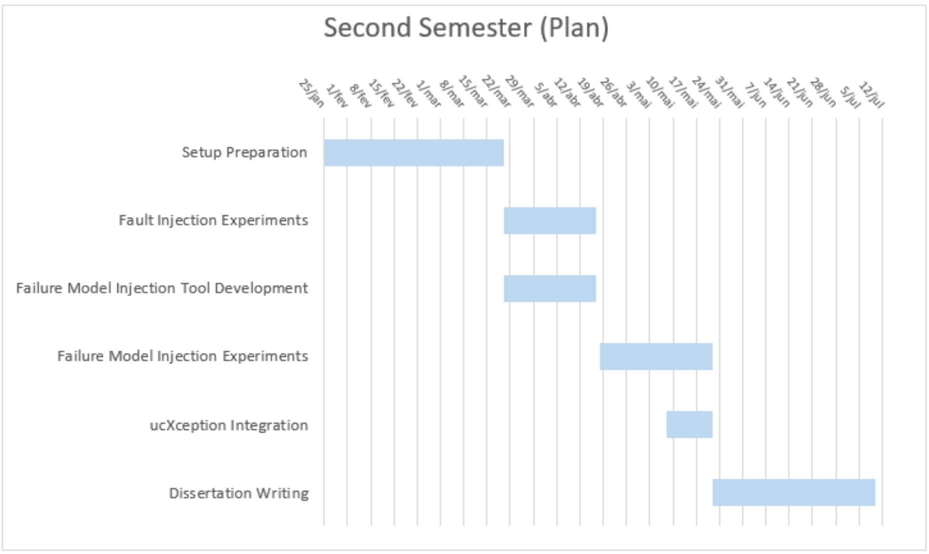


Figure 6.2: Second Semester Work

However, the actual work stages suffered some modification due to some problems, such as defects in experiments or new tasks like paper writing. The actual Gantt chart with all the task and their durations can be seen in Figure 6.3.

Since there were some risks associated with the research in the first semester, we defined the main risks that could be raised in the second semester of work and their mitigation plans. Table 6.1 defines the main risks associated with the work stages.

In the table, there are two attributes:

- The probability of occurring: low (<40%), medium (between 40% and 70%), or high (>70%).

- The impact of the risk: low (no significant difficulties), medium (may compromise a part of the work), and high (May compromises the work).

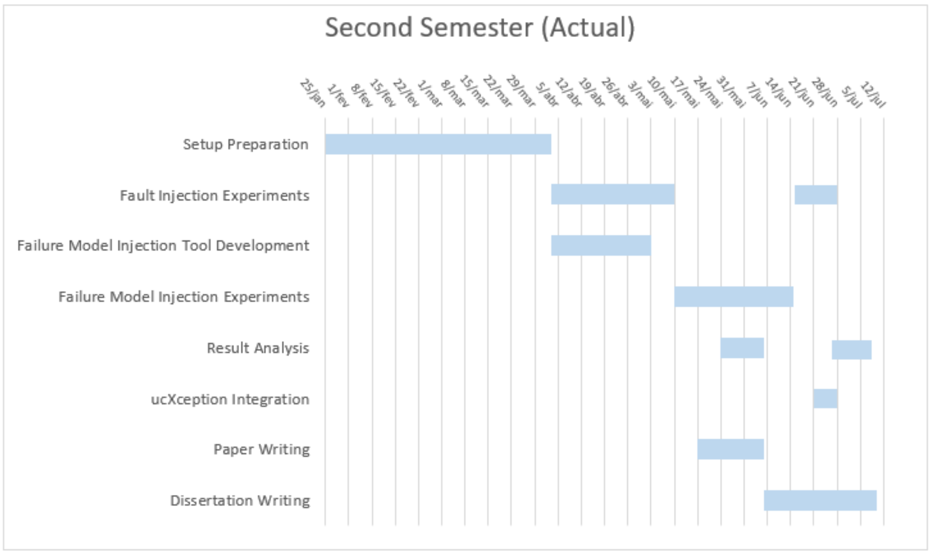


Figure 6.3: Actual Second Semester Work

ID	Risk	Impact (Severity)	Probability	Mitigation plan
R1	The build of the setup phase may be longer than expected because of the system's complexity.	Medium	High	Since it was designed and implemented initially by the advisor Frederico Cerveira, it is possible to ask him for help.
R2	FI experiments may take longer than anticipated because they can produce unreliable results.	Medium	Medium	Test initially the FI experiments process with a small number of experiments to check if the results are consistent.
R3	Delay in the development of our technique due to unfamiliarity with technologies or environments.	High	Medium	Ask for the help of advisor Frederico Cerveira.
R4	Detailed unawareness of the ucXception framework.	Low	Low	Ask for the help of advisor Frederico Cerveira since he was one of the framework's developers.

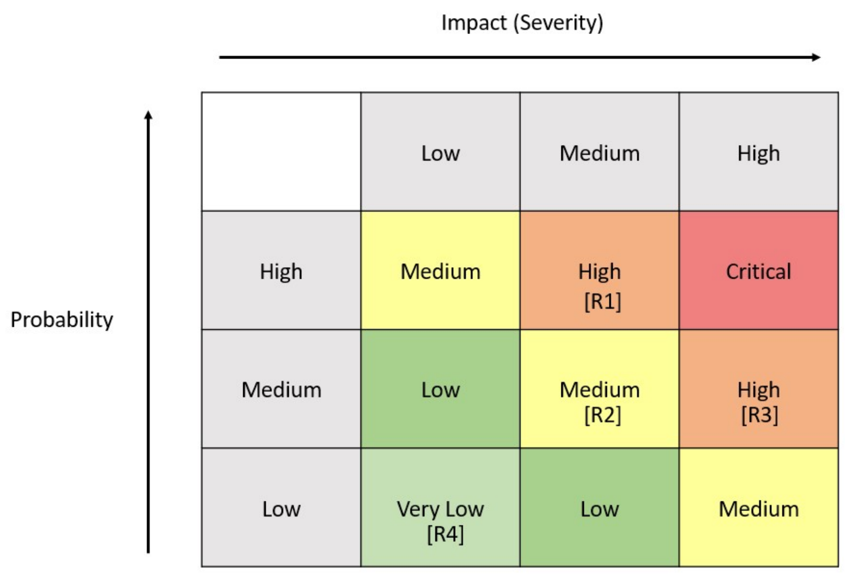
Table 6.1: Risks Table

For a complete analysis of the risks, a risk matrix is presented that can help prioritize and understand the different levels of severity of one risk. Furthermore, it allows us to analyze which risks are the most severe and that we should try to prevent from occurring. The risk matrix comprises five categories: Very Low, Low, Medium, High, and Critical.

6.2.1 Limitations

During the second semester, we had some problems related to the risk R2. Our experiment took longer than we expected since our L0 machine sometimes crashed during the experiments, thus making it necessary to stop the campaign, save the data so far, restart L0, and resume the campaign. Making the campaigns take more time due to the time lost between the crash detection and the resuming of the campaign. This problem only affected the time needed for the experiments and did not affect the results obtained.

Another problem we faced was that the first FI experiments had a defect at the


Impact (Severity)
Probability

	Low	Medium	High
High	Medium	High [R1]	Critical
Medium	Low	Medium [R2]	High [R3]
Low	Very Low [R4]	Low	Medium

Figure 6.4: Risks matrix

start of the Solr service in VMs. Thus, those experiments had to be discarded. Therefore, as mentioned, we ran new experiments of hardware and software FI where this defect was fixed. This issue is also related to risk R2, and the risk mitigation plan has been applied, leading to its resolution.

These two problems were the main limitation of our study. Since the experiments took longer than we expected, we decided to reduce our failure model to only the crash and hang, which are the main failures of virtualized systems, and thus make the comparison with the traditional hardware and software FI so that we could have enough data for a reliable and consistent comparison.

Chapter 7

Conclusion

The validation of software and hardware systems is becoming indispensable since systems are evolving, i.e., being more complex and involving multiple components. Dependability implies that service failures are less frequent than what is acceptable, and it defines multiple attributes that should exist in a system, such as availability, reliability, safety, integrity, and maintainability. Multiple threats can affect the correct service of a system. A threat can be a fault, for example, a bug in the software code or a defect in the hardware component that composes a system. When a fault is activated, it turns into an error in a component that may propagate to other dependent components and lead to the system's failure, which means that the service affected where the users or other systems that depend on that service are unable to use it.

There are multiple techniques to evaluate the dependability of a system. One of the approaches is fault injection which consists of the injection of faults in a system in order to study its behavior and mechanisms. FI can be used for software and hardware faults, and there are already multiple tools that provide FI techniques, such as ucXception.

FI accelerates the process of failures compared to their natural occurrence during system execution. Even so, one of the problems with FI is its efficiency since it can last months and be excessively intrusive to the system. Therefore, there are techniques aiming to improve FI efficiency and accelerate FI experiments. We presented multiple techniques that accelerate experiments in different manners and with different goals, such as time reduction or maximization of discovered failures.

In this dissertation, we aimed to develop a technique for FI acceleration. More specifically, we developed a FI acceleration technique that injects failure models instead of faults. Our approach accelerates experiments that aim to evaluate the dependability of a system and its fault tolerance mechanisms since it injects the effect of the faults directly. We experimentally validated our approach in a virtualized system that uses a fault tolerance mechanism (Romulus). We compared the results obtained from the failure model injection with those obtained from the traditional FI.

From this work, the failure model injection can be a valid alternative for the evaluation of a fault tolerance mechanism in some scenarios. Our main result is that the injection of crashes in L1 Hypervisor had similar results to hardware FI in all the metrics of comparison, and the injection of crashes produces results failures approximately 3x faster than hardware FI where failure percentage was about 29%. The downtime metric does not vary with the injection, thus being possible to use the failure model as an alternative to obtaining failure results. Our comparison also indicated that software FI does not produce similar times to any failure model injection. Therefore, we cannot state that the failure model injection can replace it.

FI are becoming an essential tool for the evaluation of systems and their mechanisms. The acceleration of FI experiments has been relatively underexplored so far. However, it becomes essential as systems become more and more complex, which implies the need for new techniques that are efficient and valid for their evaluation. Thus, our work intended to validate the failure model injection, which is a possible and used technique for the acceleration of FI experiments.

7.1 Future Work

This dissertation raises some questions leading to possible future work. Our comparison should be extended to other types of setups and fault tolerance mechanisms to obtain more complete results on the effectiveness and performance of failure model injection. Another possible work is exploring other types of failures that can occur in a system. Finally, the validation could encompass more experiments of both FI and failure model injection, although we have a reasonable amount of experiments for a robust comparison.

References

[1] Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson. Light-weight techniques for improving the controllability and efficiency of isa-level fault injection tools. In *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 68–77. IEEE, 2017.

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

[3] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12. Los Angeles, CA, USA, 1986.

[4] Robert Arno, Addam Friedl, Peter Gross, and Robert J Schuerger. Reliability of data centers by tier classification. *IEEE Transactions on Industry Applications*, 48(2):777–783, 2011.

[5] Shubhendu S Mukherjee, Joel Emer, and Steven K Reinhardt. The soft error problem: An architectural perspective. In *11th International Symposium on High-Performance Computer Architecture*, pages 243–247. IEEE, 2005.

[6] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco, et al. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1(2):171–186, 2004.

[7] David Oppenheimer, Archana Ganapathi, and David A Patterson. Why do internet services fail, and what can be done about it? In *4th Usenix Symposium on Internet Technologies and Systems (USITS 03)*, 2003.

[8] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, volume 1, pages 3–9, 1978.

[9] Raul Barbosa, Johan Karlsson, Henrique Madeira, and Marco Vieira. Fault injection. In *Resilience Assessment and Evaluation of Computing Systems*, pages 263–281. Springer, 2012.

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

[11] Ali Sedaghatbaf, Mehrdad Moradi, Jaafar Almasizadeh, Behrooz Sangchoolie, Bert Van Acker, and Joachim Denil. Delfase: A deep learning

method for fault space exploration. In *2022 18th European Dependable Computing Conference (EDCC)*, pages 57–64. IEEE, 2022.

[12] Roberto Natella, Domenico Cotroneo, Joao A Duraes, and Henrique S Madeira. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96, 2012.

[13] Roberto Natella, Domenico Cotroneo, and Henrique S Madeira. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)*, 48(3):1–55, 2016.

[14] Rickard Svenningsson, Henrik Eriksson, Jonny Vinter, and Martin Törngren. Model-implemented fault injection for hardware fault simulation. In *2010 Workshop on Model-Driven Engineering, Verification, and Validation*, pages 31–36. IEEE, 2010.

[15] Seungjae Han, Kang G Shin, and Harold A Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, pages 204–213. IEEE, 1995.

[16] Maha Kooli and Giorgio Di Natale. A survey on simulation-based fault injection tools for complex systems. In *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. IEEE, 2014.

[17] Jonny Vinter. An overview of goofi-a generic object-oriented fault injection framework. 2005.

[18] Joao Carreira, Henrique Madeira, João Gabriel Silva, et al. Xception: Software fault injection and monitoring in processor functional units. *Dependable Computing and Fault Tolerant Systems*, 10:245–266, 1998.

[19] Joao A Duraes and Henrique S Madeira. Emulation of software faults: A field data study and a practical approach. *Ieee transactions on software engineering*, 32(11):849–867, 2006.

[20] Frederico Cerveira, Raul Barbosa, and Henrique Madeira. Mitigating virtualization failures through migration to a co-located hypervisor. *IEEE Access*, 9:105255–105269, 2021.

[21] Pedro David Almeida, Frederico Cerveira, Raul Barbosa, and Henrique Madeira. ucxception: A framework for evaluating dependability of software systems. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 561–570. IEEE, 2022.

[22] Behrooz Sangchoolie, Fatemeh Ayatolahi, Roger Johansson, and Johan Karlsson. A comparison of inject-on-read and inject-on-write in isa-level fault injection. In *2015 11th European Dependable Computing Conference (EDCC)*, pages 178–189. IEEE, 2015.

[23] Alfredo Benso, Maurizio Rebaudengo, Leonardo Impagliazzo, and Pietro Marmo. Fault-list collapsing for fault-injection experiments. In *Annual Reliability and Maintainability Symposium. 1998 Proceedings. International Symposium on Product Quality and Integrity*, pages 383–388. IEEE, 1998.

[24] Erik Van Der Kouwe and Andrew S Tanenbaum. Hsfi: Accurate fault injection scalable to large code bases. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 144–155. IEEE, 2016.

[25] Stefan Winter, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. No pain, no gain? the utility of parallel fault injections. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 494–505. IEEE, 2015.

[26] Pallavi Joshi, Haryadi S Gunawi, and Koushik Sen. Prefail: A programmable tool for multiple-failure injection. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 171–188, 2011.

[27] Francisco Gortazár, Micael Gallego, Boni García, Giuseppe Antonio Carella, Michael Pauls, and Ilie-Daniel Gheorghe-Pop. Elastest—an open source project for testing distributed applications with failure injection. In *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–2. IEEE, 2017.

[28] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[29] Jesse Robbins, Kripa Krishnan, John Allspaw, and Thomas A Limoncelli. Resilience engineering: Learning to embrace failure: A discussion with jesse robbins, kripa krishnan, john allspaw, and tom limoncelli. *Queue*, 10(9):20–28, 2012.

[30] H. Nakama. Inside azure search: Chaos engineering. Blog, July 2015. Microsoft.

[31] Frederico Cerveira, Raul Barbosa, and Henrique Madeira. Experience report: On the impact of software faults in the privileged virtual machine. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 136–145. IEEE, 2017.

[32] Frederico Cerveira, Raul Barbosa, Henrique Madeira, and Filipe Araújo. The effects of soft errors and mitigation strategies for virtualization servers. *IEEE Transactions on Cloud Computing*, 2020.

[33] Xin Xu and H Howie Huang. On soft error reliability of virtualization infrastructure. *IEEE Transactions on Computers*, 65(12):3727–3739, 2016.

[34] IBM Support. Forcing a fake kernel panic for testing, 2021. Accessed: June 22, 2023.

[35] Stack Overflow. An alternative for tasklist$_lockinamodule$, 2012. *Accessed* : *June*22, 2023.

[36] Frederico Cerveira, Jomar Domingos, Raul Barbosa, and Henrique Madeira. Measuring lead times for failure prediction. In *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 1–5. IEEE, 2021.

Appendices

Appendix A

Fault and Failure Model Injection Flow

Appendix A contains diagrams illustrating a typical hardware and software FI experiment flow, as well as the flow for injecting failure models.

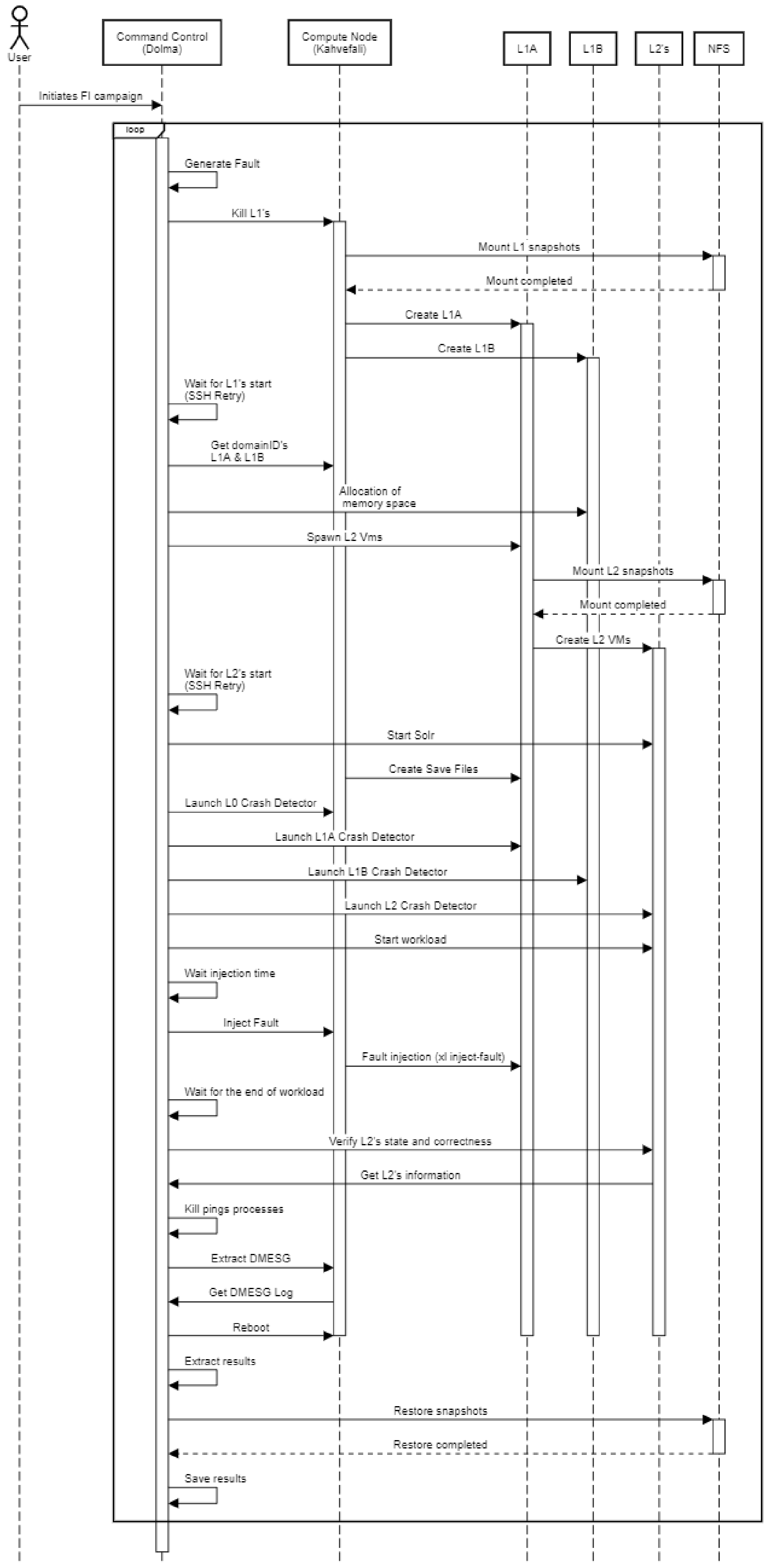


Figure A.1: Hardware FI Flow

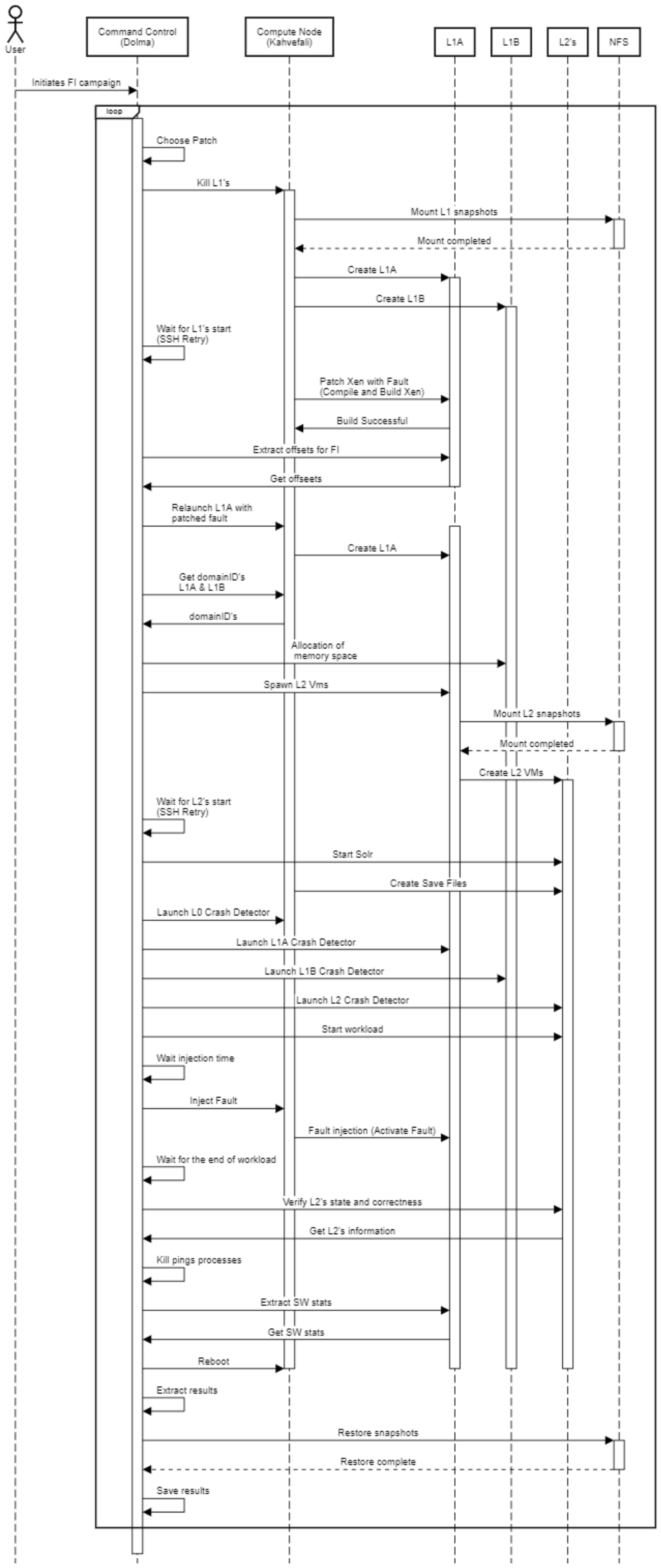


Figure A.2: Software FI Flow

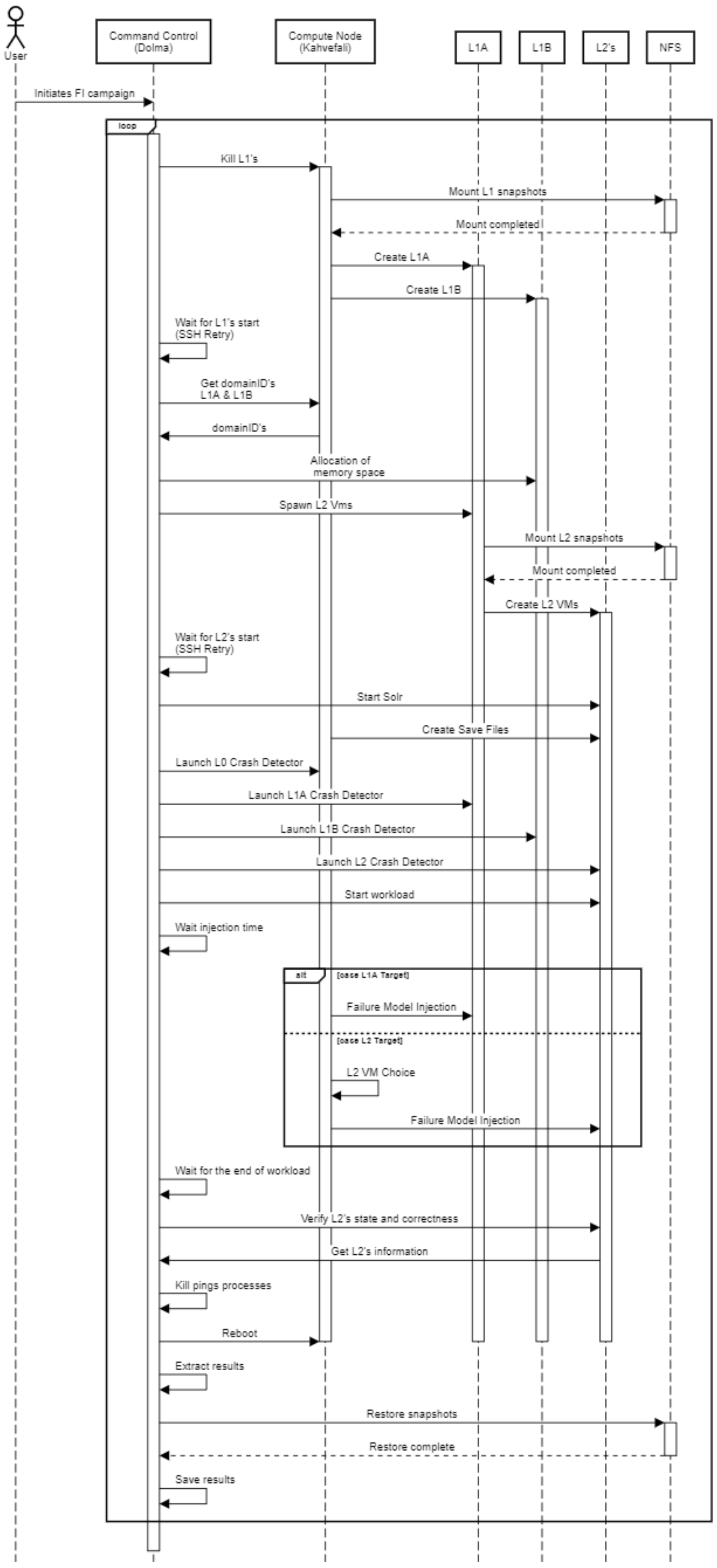


Figure A.3: Failure Model Injection Flow

Appendix B

ISSRE 2023 - Paper Submission

Appendix D contains the paper submitted to the ISSRE conference about the validation work performed in this dissertation.

On fault injection acceleration with failure models

(PER)

Abstract—**Fault Injection (FI) is an experimental technique used to evaluate system behavior under faults. It produces a significant speed-up over natural fault occurrence, condensing years of real-world operation into a few weeks. Nevertheless, FI is still time-consuming as there is a large number of parameters to test, e.g., injection location, time, and fault model, and the combination of all parameters creates a large domain space. Injecting failures instead of faults, i.e., directly inject the outcome of the fault and not just the fault or the error, is an interesting possibility to accelerate FI. This approach reduces the time required to carry out campaigns and is often used by companies to test their error detection and recovery mechanisms, however there is little hard-data about its representativeness. This paper conducts a comparison between FI using fault models and using failure models over a virtualized setup equipped with a fault tolerance mechanism. The results show that injecting failures can be an effective approach to evaluate the performance of fault tolerance mechanisms, however it is not a good alternative for evaluating the dependability of a system or for designing fault tolerance and failure prediction mechanisms.**

Index Terms—**Fault Injection, Failure Injection, Chaos Engineering, Failure Model, Dependability, Hardware Faults, Software Faults, Fault Tolerance, Virtualization, Cloud Computing**

I. Introduction

The proliferation of computer systems in supporting roles essential to everyday life, such as transportation, healthcare or communications, requires that their correct and safe operation is ensured in every situation. Although it may be relatively straightforward to verify correct behaviour when the system is operating under normal and expected conditions, it is significantly harder to verify system behaviour when in adverse conditions.

Hardware and the software that executes on it face a range of challenges that represent a threat to their dependable, safe and correct operation. Hardware can be affected by adverse physical conditions (e.g., high temperatures, vibrations, corrosion, electromagnetic interference) and aging-related problems [1]. A commonly studied problem are cosmic rays and other types of terrestrial factors, which cause transient hardware faults that appear as bit-flips in the state holding components of a computer system (e.g., the CPU registers, the cache or the memory) [2]–[4]. Software also has its problems. These are the software faults, commonly known as software bugs, of which no software component is immune, despite decades of intense research into the areas of software testing and similar [5], [6].

Fault injection (FI) is a technique for experimentally verifying the behaviour of systems when under the effect of faults [7]. It consists in the emulation of the effect of these faults, thus bringing a significant speed-up when compared to natural fault activation (i.e., operating the system as normally and waiting for faults to naturally occur).

Although FI greatly reduces the time needed to obtain failure data, it is nevertheless a time-consuming process. There is a modest amount of parameters that need to be covered during a FI campaign (i.e., a collection of FI runs planned with a common goal), which can often lead to domain space explosion. For example, in a campaign that has the objective of studying how a system behaves when affected by a cosmic ray that hits the CPU, there is the need to cover all the possible CPU registers, plus their bits and also the temporal moment when the cosmic ray hits the CPU.

Several works have strived to optimize the FI process in order to reduce the time required to conduct campaigns or to maximize the amount of failure data that is collected [8]–[11]. Many approaches are based around pruning the domain space by discarding parameter combinations that are redundant (e.g., their outcome is known *a priori*) [8].

Another approach for reducing the cost of FI consists in directly injecting the outcome of faults instead of emulating the fault itself. To this practice, we will refer to as *fault injection using failure models*, in opposition to 'traditional' fault injection, which we will refer to as *fault injection using fault models*. Many companies use fault injection using failure models (or failure injection) to test the error detection, recovery and handling mechanisms of their production systems. This practice was popularized by Netflix and its Chaos Monkey, which was then incorporated into the concept of chaos engineering [12] and adopted by the industry.

The rationale behind using failure models as an optimizer of fault injection is twofold. Firstly, it becomes guaranteed that all injections will cause a failure. In 'traditional' fault injection, a large part of the injected faults do not cause failures (i.e., they are masked), which means that it will take longer to collect failure data. Secondly, the time taken for a fault to propagate into a failure (i.e., the manifestation latency) is removed. The big disadvantage of using failure models is that the accuracy of the results is hard to assess.

This paper aims to understand whether fault injection using failure models is a valid replacement to fault injection using fault models, particularly when the objective of using fault injection is to evaluate a fault tolerance or error recovery mechanism. We perform a comparison between the two techniques using an experimental approach. Campaigns are carried out in a virtualized system using a workload that can be commonly found in cloud computing deployments. Furthermore, a fault tolerance mechanism that recovers VMs after an hypervisor failure has been installed in the system and is used to provide data about the impact that using the two different techniques can have on its performance.

The novelty of this work resides in being one of the first to verify whether the results generated by FI using failure models are comparable to those produced by fault injection using fault models. In this work, fault injection using fault models is treated as the oracle for correctness (i.e., the baseline), since plenty of works have already focused on the comparison between fault injection and natural faults [13], [14].

The results show that the injection of failures, specifically crash failures, is a valid alternative to the injection of hardware faults when the purpose of the evaluation is to measure the performance of a fault tolerance mechanism. In this scenario, the usage of failure injection can provide a speed-up close to 3x, while producing similar results. However, failure injection should not be used when the evaluation has a different objective (e.g., to collect failure data for creating failure prediction models) or to obtain results similar to those generated by the injection of software faults.

This document is structured as follows. In Section II a brief introduction to the topics of fault and failure injection is provided. Section III describes the setup used in our experimental evaluation. Section IV contains the results of our work. Section V addresses the limitations of our work. Finally, Section VI contains the conclusion and future work.

II. Related Work

This section addresses the existing approaches for accelerating the process of FI and the related work on injection of failure models. It is divided into approaches that can be applied to the injection of hardware faults and approaches aimed at the injection of software faults. At the end of this section, an overview of FI using failure models is provided.

A. Accelerating injection of hardware faults

Knowledge about the system and workload that will be targeted by FI is an important source of information whenever attempting to optimize FI. Pre-injection techniques consist of a group of techniques that analyze the target system and its characteristics before commencing the injection process. Sangchoolie et al. [8] presented two pre-injection techniques for improving controllability and efficiency in Instruction Set Architecture (ISA) level FI (i.e., injection in CPU registers and memory locations reachable from software). The first technique is Data Type Identification (DTI), used to improve the control of the time and location of the injection by identifying the data-items (the register or memory content, which can be a memory address, control information, or a data variable). The identification can be made by Instruction-Based DTI, where the machine instruction used to access some location is used to identify the data-items type, or by Location-Based DTI, where the location of the data is sued to assert the data-item type (because certain locations always hold a specific data-item type). The second pre-injection technique is Fault Space Optimization, which is employed to enhance the efficiency of a FI campaign, and it involves eliminating the bits of data-items that consistently result in an exception during system execution thus, we can remove some of the possible locations for the injection. This approach depends on intimate knowledge about the target system and the behaviour that certain bit locations have, thus it may not always be feasible to use.

Two other types of optimizations for ISA-level FI are inject-on-read and inject-on-write [9]. Inject-on-read performs the injection immediately before the reading of the respective data-item. This technique has two variants: unweighted inject-on-read and weighted inject-on-read. In unweighted inject-on-read, all faults targeting the same location are considered equally significant. However, this approach may yield unrealistic results as specific locations may be more vulnerable to faults due to containing more instructions or being more activated during the execution. Weighted inject-on-read handles this issue by assigning a weight to each location based on the probability of a fault occurrence. This weight ensures that faults with higher chances have a more significant impact on the analysis. In contrast, inject-on-write performs the injection only when the data-item is updated into a register or a memory word. The optimization that both of these techniques cause, when compared to a blind fault injection, is that faults only need to be injected in operations and locations that are used. In other words, if a certain location (e.g., a certain CPU register, a specific bit of a certain register, some memory location) is never accessed, then there is no need to inject faults in it.

Alfredo Benso et al. [10] defined a set of collapsing rules that aims to reduce the fault list size and time by analyzing the assembly code and evaluating the behavior of a system run with no faults.

Injection using failure models as an approach for accelerating FI was previously researched by P. Almeida et al. [15]. Almeida's work injected crashes of a certain type of processes of OpenStack (a famous cloud management platform) and compared the outcome against experiments that use the well-know single bit-flip in a CPU register fault model. However, their experimentation had some drawbacks, such as only evaluating one failure model (process crash), limited amount of experiment runs, and a limited setup, thus the results may not be very representative.

B. Accelerating injection of software faults

In the field of the acceleration of software fault injection, Erik van der Kouwe et al. [16] suggest a new approach called Hybrid Software FI (HSFI). This technique consists of introducing all the faults in the system in a disabled mode. Then, it uses the source-level context information to enable and disable the introduced faults. In this way, the rebuilding of the system is not necessary. Instead, two versions of the code for each injected fault are generated, a pristine version without the fault and a faulty version. In short, the idea is to have all the faults in an executable file and thus avoid the recompilation of the code multiple times, which is a very time-consuming process. Although this technique may accelerate the experiments, the representativeness of the system drops drastically because the system is heavily modified and far from the original one, as now it has all the faults embedded.

Robert Natella et al. [11] argue that the most important type of software faults are residual faults, i.e., those faults that escape testing. To this end, they suggest executing a test suite after applying a software fault but before executing the workload. If the test suite detects the injected software fault, then it is not a residual fault (it would have been caught during the normal development process) and thus should be discarded, otherwise the workload should be executed. They also propose two artificial intelligence (AI) classification algorithms aiming to inject the most representative faults selectively. The first one is based on decision trees, a supervised classifier trained using examples of components. The second algorithm employs k-means clustering, an unsupervised classifier that identifies that depends on observing the component less exposed to testing. By prioritizing the most representative faults first, it is possible to obtain more failure data and converge faster towards the correct result in a fewer number of runs.

Stefan Winter et al. [17] proposed an approach called "Parallel Fault Injection" (PAIN), which is based around the idea of parallelizing the execution of experiments, thus maximizing the available computing resources. However, this technique depends on a crucial assumption, which is that the execution of multiple experiments at the same time will not affect the results. Unfortunately, their own experiments concluded that their approach leads to a degradation of the representativeness of the results, depending on the degree of parallelism that is used.

The field of artificial intelligence has also been explored to accelerate FI campaigns. Ali Sedaghatbad et al. [18] proposed a deep learning method for fault space exploration called DELFASE. It consists of Generative Adversarial Networks (GAN) for the identification of critical faults (faults that reveal system vulnerabilities leading to the violation of safety requirements). The goal is to make the generator capable of generating dummy data that the discriminator cannot distinguish from the real one to be able to predict the behavior of a failure. This technique has two modes: an active, where a ranked batch-mode sampling is used to select faults for the GAN model training, and a passive, where the faults are selected randomly. Thus, DELFASE accelerates the FI experiments by selecting the critical faults through the use of active learning and GAN resulting in a higher fault coverage.

C. FI using failure models

Failure injection is already used in various companies to test their systems and recovery mechanism behavior when a failure occurs. These techniques are named Chaos of Engineering. Netflix uses Chaos Monkey to test the availability of its video stream services for the client [12]. After Netflix and due to the increased need to ensure service resilience, several companies have also adopted Chaos Engineering techniques, such as Google and Amazon, which designed a program called GameDay to find bugs and defects in their system. This program consisted in injecting failures in the critical systems [19]. Microsoft, inspired by Netflix Chaos Monkey, uses Azure Search Chaos for the controlled inject controlled disruptions and to simulate failures in a test environment in order to identify the weaknesses and defects [20].

There is also some research on failure injection, Pallavi Joshi et al. proposed PreFail, a programmable failure-injection tool with the goal of avoiding the combinatorial explosion of experiments. PreFail enables the user to set a range of policies to reduce the vast space of multiple failures, thus reducing the number of needed experiments [21]. Also, Francisco Gortázar et al. introduced ElasTest, a platform that performs failure injection intending to simplify the testing of distributed applications. This open-source project allows the injection failures like packet loss, CPU bursting, and node failure, among others [22].

These tools already make use of the failure injection in order not just to accelerate the experiments but also to test the systems in the occurrence of specific types of malfunctions.

III. Experimental Setup

The experimental setup is a key piece in our study between the injection of fault and failure models. The chosen setup is based on a virtualized system hosting multiple Virtual Machines (VMs) that migrate from one hypervisor to another co-located hypervisor whenever one fails through a technique called Romulus, designed to tolerate hypervisor failures. The inclusion of a fault tolerance mechanism means that we can evaluate the impact that injecting failure models can have on its performance. This section describes the used experimental setup, workload, fault and failure models and the metrics chosen to measure the results.

A. Physical Setup

The physical setup comprises two nodes: i) a Compute Node hosting the hypervisors and the VMs; and ii) the Orchestrator Node, responsible for the experiments management and the provisioning of disk images to the VMs through a network file systems (NFS). Figure 1 systematizes the two nodes and their composition.

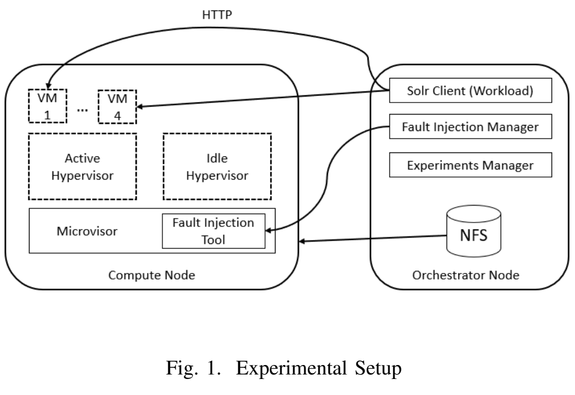


Fig. 1. Experimental Setup

The Compute node is composed of two Intel Xeon Silver 4114 with ten physical cores each, 32 Gb of RAM and a network interface that can support a maximum speed of 1 GbE. The Orchestrator node has a single Intel Xeon E5620 processor with four physical cores, 12 Gb of RAM, and a network interface supporting a maximum speed of 1 GbE.

B. Virtualized Setup

In the Compute node, the first layer (or L0) contains a microvisor, i.e., a small hypervisor, that is a requirement of the installed fault tolerance mechanism. The microvisor enables running two hypervisors on the same physical hardware, through a technique known as nested virtualization. Thus, in the second layer (L1), we have two Xen hypervisors. One of the two hypervisors is active and supporting the VMs (we call it L1A), while the other is idle and plays the role of a 'backup' hypervisor (we call this hypervisor L1B). Finally, in the last layer (L2), we find the VMs that support the workload. For these experiments, we configured the system with a total of 4 L2 VMs. Figure 2 presents the different software layers and the composition of the virtualized system.

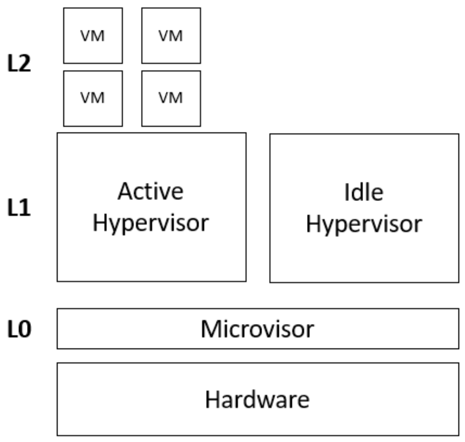


Fig. 2. Virtualized System Layers

C. Romulus

The fault tolerance mechanism that is installed in the system has the objective of protecting the L2 VMs from failures of the hypervisor (L1A) through their migration from the active hypervisor (L1A) to the backup hypervisor (L1B). Of course, after the migration process is finished, the backup hypervisor becomes the new active hypervisor. This fault tolerance mechanism is called Romulus.

The approach used by Romulus is versatile and adaptable to multiple hardware architectures and to different hypervisors. However, it has some requirements. First of all, a microvisor has to be added to the virtualization system in order to manage the migration process. Furthermore, nested virtualization has to be used to support executing two L1 hypervisors side-by-side and virtual machine introspection (VMI) has to be supported. The final requirement is that hardware-assisted virtualization is supported by the system and used by the L2 VMs.

Before VM migration can take place, several actions must be taken. First, it is necessary to pre-allocate memory space for holding the VMs on the backup hypervisor (L1B). Then, a template save file containing the information about each L2 VM is created. Finally, the VMCS data structure of each L2 VM is monitored and kept up-to-date by the microvisor.

After all the above steps have been taken, a failure detection mechanism is launched. This mechanism is independent of Romulus, thereby giving freedom to the user to choose the mechanism that best fits a scenario. In our experiments, we used a simple mechanism, which consists of monitoring the active hypervisor and the L2 VMs with ping requests. After a certain amount of ping requests fail, we consider that a failure has occurred and Romulus is triggered, thus beginning the migration process.

In the migration process, firstly, the L1A is paused to extract the CPU and I/O state of the L2 VMs. Then the template save files are updated with the latest state. The memory state of every L2 VM is migrated from the active hypervisor (L1A) to the other hypervisor (L1B). Finally, the L2 VMs are restored using their save files in the L1B.

D. Workload

In our experiments, we defined a workload that emulates a typical Solr client-server scenario. Solr is an open-source platform used for text-based searches. Each L2 VM hosts a Solr server that is filled with a part (9Gb) of Wikipedia's index, while multiple clients can be emulated in the Orchestrator node. Our workload is set to have one client making a request each second to the Solr servers in each experiment.

We configured the workload to last either 600 seconds or 1200 seconds, depending on the type of fault that is being injected. The difference in length of the workload arises because certain types of faults (namely, software faults) have a longer manifestation latency.

The correctness of the provided responses is verified by comparing the obtained output against a pre-calculated expected value. In this manner it is possible to detect both requests that were not answered and requests that were answered incorrectly (i.e., silent data corruption).

E. Fault Models

Two different types of faults were considered in our experiments: transient hardware faults (soft errors) and software faults.

To emulate transient hardware faults, we used the single bit-flip fault model. In each experiment run, a random register and bit was flipped. The considered registers were RIP, RSP, RBP, RAX, RBX, RCX, RDX and R8 to R15. Faults affecting memory were not considered for two main reasons. Firstly, the memory of server-grade deployments is usually protected with effective error-correcting codes [23]. Secondly, faults affecting memory can be emulated using bit-flips in CPU registers. This approach can be considered a naive approach because we do not perform any steps to optimize the FI process (i.e., we are not using knowledge about the system or workload to filter the domain space).

To emulate software faults, the chosen fault model was the one proposed by Durães et al. [24]. Out of the 13 operators in the short-list, we inject 10: WVAV, WPFV, WAEP, MVAV, MVAE, MLAC, MIFS, MIEB, MIA, and MFC. Another two operators (WLEC and WALR) are used and obtained from [25]. At the beginning of each experiment run, the software fault is introduced in the code of the hypervisor in a disabled state. During the experiment run, after a certain amount of time has been elapsed, the fault is enabled. After the fault is enabled, it may then be activated (i.e., executed). Faults were injected in 4 specific source-code files, which are "arch/x86/hvm/vmx/vmx.c", "arch/x86/hvm/vmx/vmcs.c", "arch/x86/msr.c", and "arch/x86/mm.c". These files were chosen because they have important yet different roles in the hypervisor and the code lines that they contain are executed by the workload (which was verified before conducting the experiments). In this case, we used an optimized FI process because we filtered out software faults that affected lines that we knew were not executed by the workload and only injected faults in lines that were executed.

F. Failure Models

The failure models that were evaluated are crash and hang. These two failure models were chosen because they have been often used in the literature and because in cloud systems, the most common failure mode is a crash [26]. A hang is not as common as a crash but can occur, making the system unresponsive and subsequently failing.

G. Injection Points

Given a relatively complex system with various layers like the one used for our experiments, it becomes feasible and even desirable to inject faults and, specially, failures in different points of the system. For that purpose we define two injection points, which are depicted in Figure 3. These injection points are in the hypervisor (more specifically, the active hypervisor) and in a L2 VM.

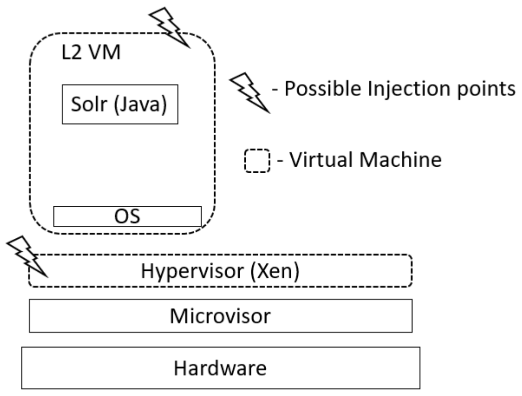


Fig. 3. System Detailed Architecture

For the injection of hardware and software faults, the used injection point was the hypervisor. This makes sense given that Romulus aims to protect against hypervisor failures, thus injecting faults in the L2 VM would be outside the coverage provided by Romulus. On the other hand, failures were injected both in the hypervisor (e.g., crashing the hypervisor) and in the L2 VMs (e.g., crashing a VM). Injecting failures in the VM is justifiable because previous studies have shown that faults in the hypervisor often cause crash and hangs of one or more VMs [26], [27].

H. Experiment Flow

Each experiment consists of a set of tasks. The first task is launching the L1A and L1B from snapshots. Then, the L2 VMs are launched and the steps required to prepare Romulus are taken (e.g., pre-allocating memory in L1B). After these preparatory steps are finished, monitoring probes are launched to detect failures in the VMs. The workload is initiated, and after a specific time (i.e., when the warmup phase has ended), a single fault or failure is injected. The workload is left to execute until the end. If the monitoring probes detect that at least 1 of the VMs had failed, the recovery process is triggered. After the workload finishes, tests are carried out to verify the state of the VMs, then the VMs are turned off, their state restored back to a pristine state and the physical machine is rebooted to avoid carrying over anything that may interfere with the next experiment run.

I. Metrics

The comparison presented in this work will focus on four different metrics. If the results obtained using FI using failure models are similar to the results of FI using fault models, then we can state that both techniques are interchangeable. The metrics are:

1) Failure percentage - The amount of failures divided by the total number of injections;
2) Manifestation latency - The time elapsed between the moment of injection and the moment when a failure is detected in a VM;
3) Recovery success - The percentage of VMs that were successfully recovered after Romulus was triggered;
4) VM downtime - The amount of time that the VM was unavailable while recovery took place.

The two first metrics focus only on the behaviour exhibited by the system after an injection is performed, whereas the other two metrics focus on the behaviour of the fault tolerance mechanism. Failure percentage is the most straightforward metric and symbolizes the speed-up in the occurrence of failures. Naive techniques usually show lower failure percentages than optimized techniques. This is not a problem when the purpose of fault injection is failure data collection (e.g., to train machine learning models), but it becomes a problem when fault injection is used to characterize the dependability of the system, since the failure percentage obtained using an optimized technique will not reflect the real failure percentage of the system.

Manifestation Latency refers to the time between the fault injection and the earliest manifestation of the effects caused by it. This metric is very important when designing fault tolerance mechanisms and failure prediction models, because it governs the amount of time that is available for these mechanisms to act. It has been shown that manifestation latency naturally varies depending on the complexity of the system and even on the type of fault [28]. Thus, for collecting failure data to be used in two aforementioned applications, it would be important for FI using failure models to produce similar manifestation latency as FI using fault models. On the other hand, high manifestation latencies can significantly reduce the amount of injections that can be performed in a certain amount of time, thus slowing down the FI campaigns. In our experiments, the observation point is the Solr Client (the point where we will detect the failure), thus our manifestation latency is calculated from the client's viewpoint. Figure 4 exemplifies the manifestation latency time in a typical fault injection.

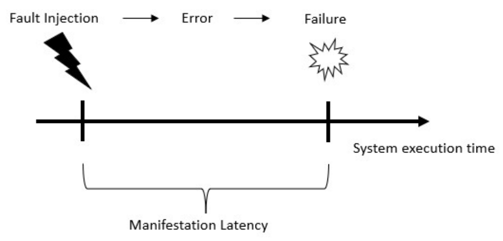


Fig. 4. Manifestation latency

Recovery success, i.e., the percentage of recovered VMs, is a key metric used for the evaluation of fault tolerance mechanisms. When a failure occurs, Romulus is triggered but it may not always successfully recover all VMs that the failed hypervisor hosted. For FI using failure models to be interchangeable with FI using fault models wrt. evaluating fault tolerance mechanisms, then it should produce similar results in this metric.

The final metric is also important for the evaluation of fault tolerance mechanisms. The VM downtime refers to how much time the VMs are unavailable as a result of the recovery process. This metric should show similar values between traditional FI and FI using failure models, otherwise FI using failure models cannot be trusted to accurately evaluate fault tolerance mechanisms.

IV. Results & Discussion

This section begins with a characterization of the various datasets that were used for this comparison, followed by a presentation and discussion of the results grouped according to each of the metrics.

A. *Dataset characterization*

In our comparison, we use eight datasets, which are detailed in Table I. Each dataset represents a cluster of experiments grouped in time and multiple datasets may use the same fault or failure model. In total we analyze 2809 runs where transient hardware faults are injected (HW_A and HW_B), 1200 runs where software faults are injected (SW_A and SW_B), 597 runs where crash failures are injected (CRASH_L1 and CRASH_L2) and 299 runs where hang failures are injected (HANG_L1 and HANG_L2). Hardware and software faults, as well as the crashes of CRASH_L1 and hangs of HANG_L1, were injected on the active hypervisor (L1_A), whereas crashes and hangs of CRASH_L2 and HANG_L2 were injected in a VM chosen at random.

TABLE I
Datasets Characterization

Dataset	Experiments
HW_A	2409
HW_B	400
SW_A	832
SW_B	368
CRASH_L1	297
CRASH_L2	300
HANG_L1	100
HANG_L2	199

For a better understanding of the datasets where transient hardware faults were injected, we look into the distribution of injected faults across CPU registers. Ideally each register would have a similar amount of runs, however, as can be seen in Figure 5, in dataset HW_A, the rip and rsp registers are overly represented, followed by rax, rbp, rcx, and rdx. While in HW_B, the different registers show a more balanced distribution.

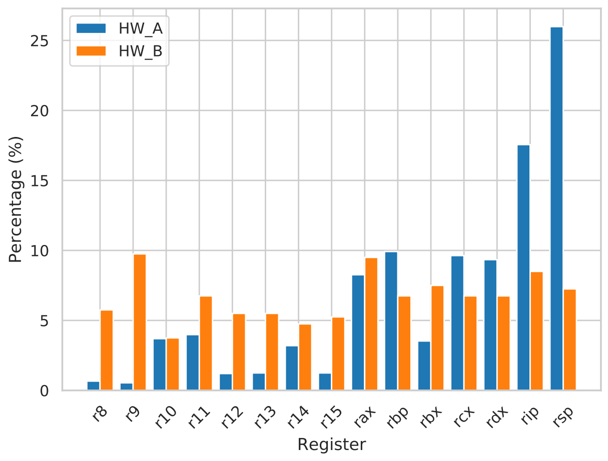


Fig. 5. Registers Distribution

A similar analysis was performed for the datasets where software faults were injected as to study which operators were more often injected. As can be seen from Figure 6, the most common operator were, by far, MFC and MIA, followed by MIFS. This is uneven distribution among operators is to be expected because operators have pre-conditions that must be true for them to be applied (e.g., the MFC operator has to be applied wherever a function call is being done) and some pre-conditions are more likely to occur than others.

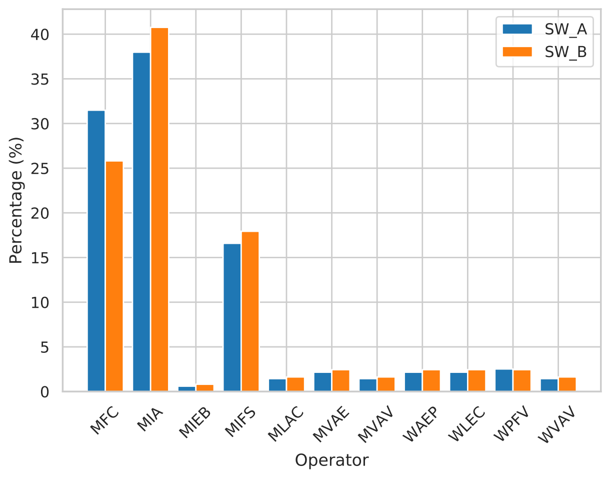


Fig. 6. Operators Distribution

B. *Analysis wrt. failure percentage*

Our first analysis focuses on the the failure percentage metric. Higher failure percentages mean that fewer experiments are required to obtain a certain amount of failures, because there are fewer runs where the injected fault was masked and did not cause a failure. Table II shows the results.

TABLE II
FAILURE STATISTICS PER DATASET

Dataset	Num. Failures	Failure Percentage (%)	Weighted Failure Percentage (%)
HW_A	1467	60.9	27.17
HW_B	124	31.00	27.86
SW_A	767	92.19	***
SW_B	328	89.13	***
CRASH_L1	297	100	***
CRASH_L2	300	100	***
HANG_L1	100	100	***
HANG_L2	199	100	***

Since the HW_A dataset has an unbalanced distribution among registers, we decided to calculate the weighted failure percentage for datasets HW_A and HW_B. This percentage is calculated by looking at the register with the highest number of injections, then extrapolating the number of injections of every other register until all registers have the same amount of injections. The number of new failures is calculated by multiplying the failure percentage of the register before extrapolation by the new number of injections. After this extrapolation process, the normal average is calculated. This process was effective because the obtained weighted failure percentage is similar for HW_A and HW_B, whereas the original failure percentage of both datasets is quite different.

SW_A and SW_B showed a relatively high failure percentage because the used fault injection process was already optimized (as stated in Section III-E). For example, we only inject software faults in lines of code that the workload executes, therefore we prune the domain space from many faults that would never lead to a failure. Since the fault injection process used for transient hardware faults was not optimized (we solely injected in a randomly chosen register and bit), there is a significant difference in failure probabilities between them.

As expected, experiments where failures were injected had perfect effectiveness. We can conclude from these results that injecting failures represents an improvement of almost 3x over naive injection of hardware faults and about 11% over an already optimized injection of software faults.

C. *Analysis wrt. manifestation latency*

The second metric that we analyze is manifestation latency, which is the time between the actual moment of injection and the first perceptible sign of failure. Manifestation latency is a key metric for designing certain fault tolerance mechanisms (e.g., rollback-based mechanisms) and failure prediction models. In this analysis, the datasets were filtered to contain only the experiences that resulted in a failure.

Figure 7 shows the manifestation latency distribution for each dataset. This time was measured using the same mechanism used to detect failures and trigger Romulus, which consists in various pings to the L2 VMs.

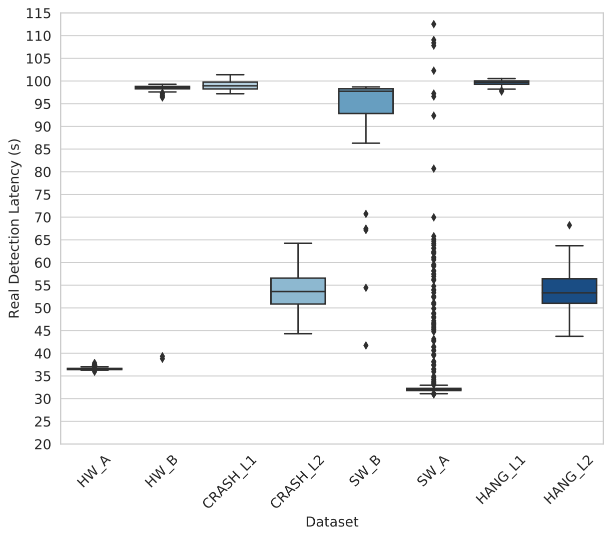


Fig. 7. Manifestation Latency - Ping

We can observe that the HW_A dataset presents the smallest values with an average of 36.75s and a median of 36.54s. The average manifestation latency of the SW_A is 63.27s. Still, the median is 32s, indicating that the experiments do not behave similarly in various situations, which explains the presence of multiple outliers. Thus the SW_A experiments are less consistent than the HW_A experiments. The consistency of the manifestation latency is also an important factor contributing to the length of an experiment run. If the manifestation latency is very inconsistent, the person designing the campaign must take the highest possible value into consideration when defining the amount of time that the workload should run. If the workload is too short, then some faults that would cause a late failure are cut abruptly and incorrectly labeled as having had no effect.

As we can observe, the HW_B and SW_B have similar values, with an average of 97.49s and 94.78s, respectively. Our results show that the CRASH_L1 and HANG_L1 experiments have a manifestation latency similar to HW_B and SW_B too, while CRASH_L2 and HANG_L2 have a lower manifestation latency. These observations were unexpected because the latency of injecting a crash failure should be very low. For that reason, we decided to calculate the manifestation latency as seen from a different source, the workload clients themselves. In other words, we measured the latency between the moment of injection and the first request that failed. The results are shown in Figure 8.

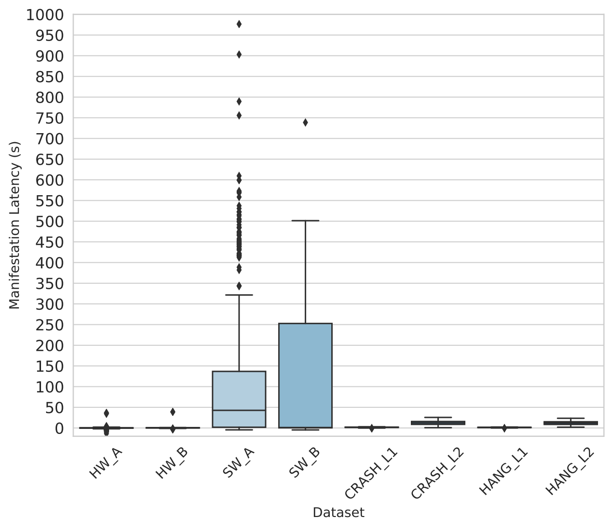


Fig. 8. Manifestation Latency - Solr (Workload)

The manifestation latencies looking at the Solr requests agree much more with our expectations for the datasets CRASH_L1 and HANG_L1. A possible explanation for the discrepancy in latencies when using ping and Solr is the effect of a network timeout mechanism, however no further inquiries were performed to confirm this suspicion.

It can be understood that the manifestation latency times of SW_A and SW_B datasets are much longer, thus software failures may take significantly longer to produce effects on the target system. To visualize in more detail the values of hardware datasets compared to the crashes and hangs, we generated a boxplot with only the failures and hardware datasets, which is depicted in Figure 9.

The median of HW_A and HW_B is 0.18s and 0.32s, respectively. These hardware datasets are similar to CRASH_L1 and HANG_L1, where the medians are 1.71s and 1.29s. HW_A, HW_B, CRASH_L1, and HANG_L1 have some negative outliers that small measurement errors can explain since our environment is a distributed system, then there are always synchronization problems associated with the different ma-

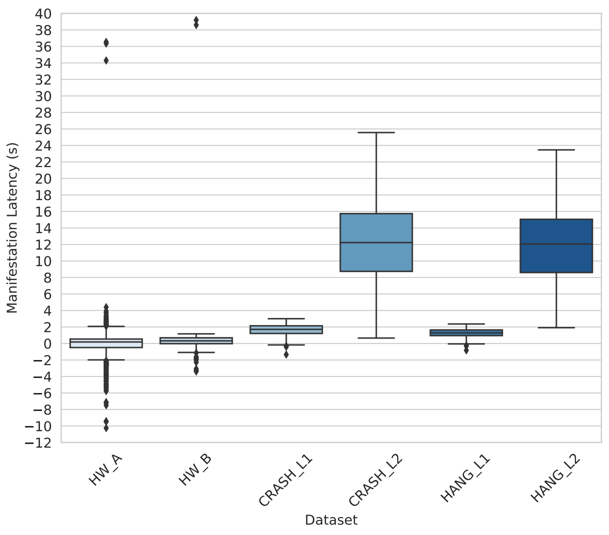


Fig. 9. Manifestation Latency (subset of datasets)

chines. The timestamps used for this calculation are retrieved from the Orchestrator node and the hypervisor, and both can present small variations. Looking at the manifestation latency times, the crash and hang of L1 produce identical times to those produced by hardware faults in both hardware datasets compared to CRASH_L2 and HANG_L2, where the median is much higher, 12.28s and 12.06s. These results allow us to conclude that injecting crash and hang failures in the L1 is a valid (and quicker) alternative to injecting hardware faults regarding manifestation latency. However, the same cannot be said of injecting these types of failures in the L2.

D. Analysis wrt. VM recovery percentage

Using a failure model in an experimental evaluation of a fault tolerance mechanism should only be done if the resulting behaviour of the mechanism is similar to when fault models are used. To verify this, we look into how effective Romulus was at recovering VMs when different faults and failures are injected. For this evaluation, we focus on the HW_A, SW_A, CRASH_L1, CRASH_L2, HANG_L1 and HANG_L2 datasets, because these had the highest amount of usable runs.

We analyze the performance of Romulus from two points of view, 1) Solr, i.e., whether there has been a recovery of the Solr service in the VMs, and 2) the operating system (OS), i.e., whether the operating system kept working correctly even if Solr did not. This differentiation exists because Romulus, in certain occasions, may be able to recover the O.S but not the Solr service. If this occurs, a simple mechanism to restart Solr could provide service continuity with low downtime and cost.

Regarding the Solr point-of-view, Figure 10 contains the bar chart of the distribution of recovered VMs. In general, the most common outcome is the recovery of two VMs, where HANG_L2 and CRASHŁ2 have the highest values (44.72% and 42.33%) followed by HW_A (36.22%), CRASH_L1 (34.45%), SW_A (24.71%) and HANG_L1 (24%). The SW_A is the only dataset where the probability of recovery of no VM is the highest, with 27.82%. CRASH_L2 and HANG_L2 cannot recover the four VMs in any experiment. This is expected because the failure involves crashing one of the L2's VMs, making its recovery always impossible.

Regarding the OS point-of-view, Figure 11 shows the probabilities of recovery of the operative system of L2 VMs. As expected, the crash of one L2 VM does not affect the other VMs performance, and therefore, the L2_CRASH has a probability of 69.3% recovery of three VMs. SW_A has a high probability of recovering all the VMs, with 44.68%. However, it also has a probability of 22.91% of not recovering any VM. It is worth noting that the hang failures present a higher percentage of recovered VMs in comparison with crashes. HANG_L1 has a 62.0% probability of recovering the four VMs, and HANG_L2 has a 72.36% probability of recovering three VMs which is the maximum when injecting a hang on an L2 VM.

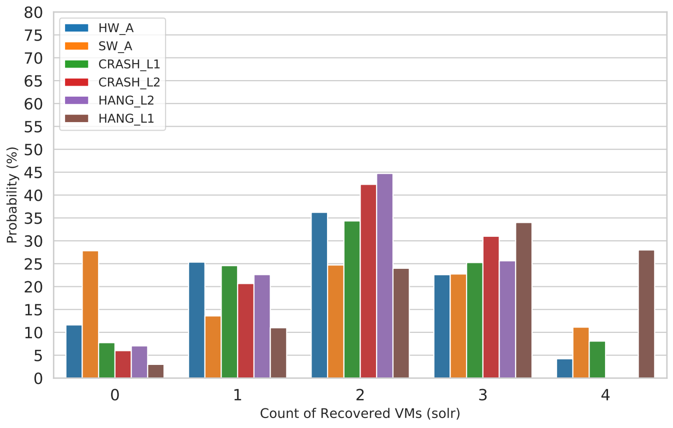


Fig. 10. Recovered VMs (Solr)

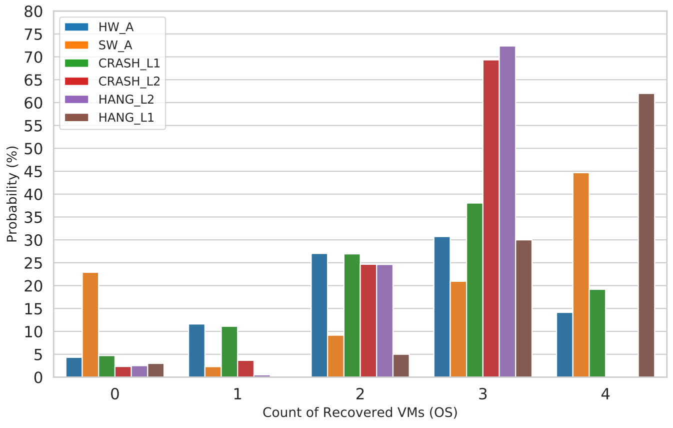


Fig. 11. Recovered VMs (OS)

To better express the results, we created a cumulative histogram of the probabilities of VM recovery. These histograms conclude that HANG_L1, CRASH_L2 and HANG_L2 are where the recovery is most successful. SW_A is where the recovery capacity is lower, although it has a reasonable probability for three or more VMs compared with the other datasets. The two histograms also show the similarity between the HW_A and CRASH_L1, causing the failure injection to be a valid way to study the recovery, as concluded above.

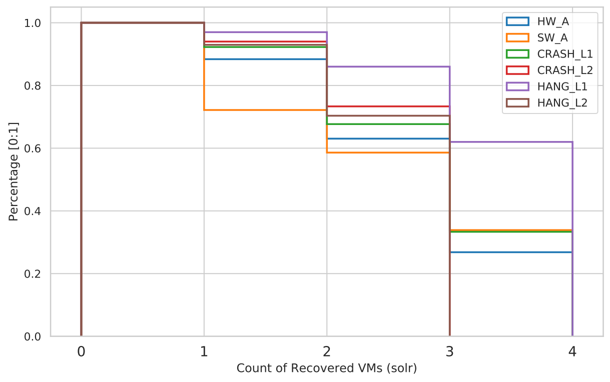


Fig. 12. Cumulative Histogram Recovered VMs (solr)

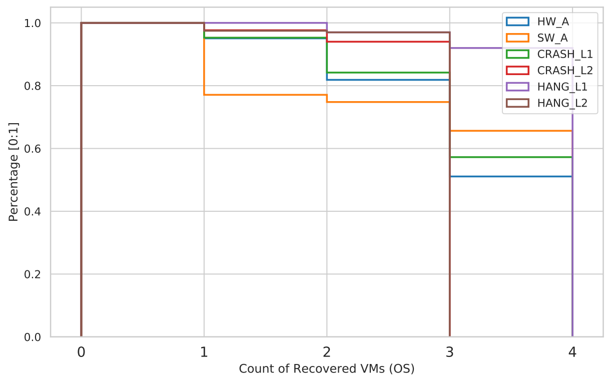


Fig. 13. Cumulative Histogram Recovered VMs (OS)

From the results, we infer that HW_A and CRASH_L1 have similar probabilities of VM recovery. This observation supports the use of the crash failure model as a valid alternative to traditional hardware FI, when the goal is to study the success of recovery of a fault tolerance mechanism. However, the user should expect the results using injection of crashes to be slightly more optimistic (i.e., better VM recovery) than when injecting hardware faults.

E. Analysis wrt. VM recovery downtime

The other metric associated to the performance of the recovery mechanism is the VM downtime. The recovery process as provided by Romulus always incurs downtime, which accounts for the time taken to migrate from one hypervisor to the other. We study whether different fault and failures models have an impact in the VM downtime.

Figure 14 presents the downtime of the multiple L2 VMs. HW_A and SW_A have a median of 285.48s and 301.21s, while CRASH_L1 and CRASH_L2 are 250.75s and 268.04s, and last HANG_L1 and HANG_L2 have a median of 270.50s

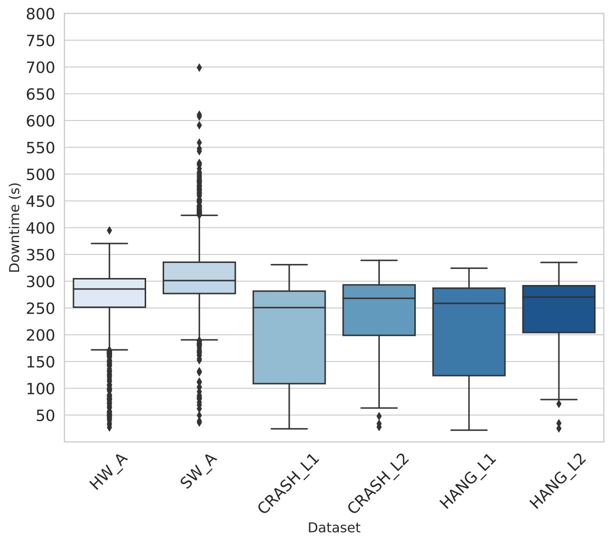


Fig. 14. Recovery Downtime

and 258.65s respectively. However, HW_A and SW_A contain more outliers, which can be explained by the unpredictability of the injection of a fault compared to the crash, which behaves almost the same way.

These results suggest that downtime does not vary with failure injection. Therefore, for the study of the downtime of VMs, both fault injection and failure injection can be considered since they produce similar values, thus not having much influence on the experiments and do not limit the analysis of it.

F. Summary

From the results, the main observations to extract are that certain failure models (namely, crash failures in L1) can produce manifestation latencies similar to those generated by fault injection of hardware faults. The same can be said regarding the VM recovery percentage of CRASH_L1 and hardware failures. At the same time, the downtime of VM recovery is widely similar between all the studied fault and failure models. From here we can conclude that the crash failure model applied to the L1 is a good alternative to fault injection of hardware faults, because it can produce similar results at a much quicker pace (about 3x quicker). However, it should be noted that CRASH_L1 appears to produce slightly more optimistic results regarding VM recovery percentage than hardware faults.

On the other hand, certain injection points (L2) and some failure models (hang) produced results that deviated more than desired and are less useful as an alternative to fault injection. Furthermore, no failure model was able to accurately emulate the results of software fault injection.

V. Limitations

As with any experimental evaluation there are factors that limit the applicability and ability to generalize the results. The first limitation that we identify in this work is the fact that only a single setup was evaluated. Furthermore this setup represents a very specific application: a node of a cloud computing deployment which hosts VMs that support a read-heavy Solr-based workload. It is possible that conducting the same kind of experiments on a different setup, or even on a different workload, can lead to different results. As future work we plan to study more setups as to reduce this limitation of our work.

The second limitation is the evaluated fault tolerance mechanism, which is not a production-ready and widely deployed mechanism. As such, it is possible that the mechanism itself can affect the results. For example, if a more mature mechanism, or even just a different mechanism, had been evaluated, the results might have been different. This limitation can only be solved by performing similar campaigns using different fault tolerance mechanisms.

The third limitation is the considered failure models. Although the two considered models are the most commonly used (crashes and hangs), various works have used different failure models, such as disk errors or memory and CPU usage. As future work we plan to consider a wider range of failure models. Furthermore, we plan to inject crashes and hangs in other injection points, e.g., inside processes of the VM.

The fourth and last limitation that we identified is the amount of experiment runs. Although we believe that we have enough runs to justifiably support our observations, a higher number of runs would always provide more certainty and trust in the results. This limitation requires time to be solved, but we are continuously conducting more experiment runs towards that objective.

VI. Conclusion

Many systems nowadays need to avoid and tolerate failures because they are used in safety-critical and highly-dependable scenarios. As such, it is necessary to study how systems behave under specific conditions, such as when affected by faults. Fault injection is an experimental approach which is very useful for evaluating the dependability of systems and validating fault tolerance and error detection mechanisms. Through the intentional injection of faults, the natural fault occurrence process is accelerated. However, fault injection techniques can be time-consuming, thus, there is interest in accelerating the process to get results faster and more efficiently. This work studies the potential of failure injection as a fault injection acceleration technique.

The results show that, in a virtualized setup with a fault tolerance mechanism (Romulus), failure injection can be an alternative to traditional fault injection when the goal is to analyze the fault tolerance mechanism's performance or to study the downtime of VMs. However, for different applications, failure injection is not an adequate replacement, and fault injection remains the right tool. In the future, we will consider more failure models and conduct more experiment runs to reinforce the confidence in our results. Different environments, workloads and fault tolerance mechanisms will also be studied.

References

[1] V. Narayanan and Y. Xie, "Reliability concerns in embedded system designs," *Computer*, vol. 39, no. 1, pp. 118–120, 2006.

[2] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sept 2005. [Online]. Available: http://dx.doi.org/10.1109/TDMR.2005.853449

[3] J. W. McPherson, "Reliability Challenges for 45Nm and Beyond," in *Proceedings of the 43rd Annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: ACM, 2006, pp. 176–181.

[4] S. Borkar, "Design Perspectives on 22Nm CMOS and Beyond," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. New York, NY, USA: ACM, 2009, pp. 93–94.

[5] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, Jan 2013.

[6] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, Aug 2000.

[7] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr 1997. [Online]. Available: http://dx.doi.org/10.1109/2.585157

[8] B. Sangchoolie, R. Johansson, and J. Karlsson, "Light-weight techniques for improving the controllability and efficiency of isa-level fault injection tools," in *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2017, pp. 68–77.

[9] B. Sangchoolie, F. Ayatolahi, R. Johansson, and J. Karlsson, "A comparison of inject-on-read and inject-on-write in isa-level fault injection," in *2015 11th European Dependable Computing Conference (EDCC)*, 2015, pp. 178–189.

[10] A. Benso, M. Rebaudengo, L. Impagliazzo, and P. Marmo, "Fault-list collapsing for fault-injection experiments," in *Annual Reliability and Maintainability Symposium. 1998 Proceedings. International Symposium on Product Quality and Integrity*, 1998, pp. 383–388.

[11] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2013.

[12] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[13] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. Leber, "Comparison of physical and software-implemented fault injection techniques," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1115–1133, 2003.

[14] H. Schirmeier and M. Breddemann, "Quantitative cross-layer evaluation of transient-fault injection techniques for algorithm comparison," in *2019 15th European Dependable Computing Conference (EDCC)*, 2019, pp. 15–22.

[15] P. David Almeida, F. Cerveira, R. Barbosa, and H. Madeira, "ucxception: A framework for evaluating dependability of software systems," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, 2022, pp. 561–570.

[16] E. Van Der Kouwe and A. S. Tanenbaum, "Hsfi: Accurate fault injection scalable to large code bases," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 144–155.

[17] S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, "No pain, no gain? the utility of parallel fault injections," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 494–505.

[18] A. Sedaghatbaf, M. Moradi, J. Almasizadeh, B. Sangchoolie, B. Van Acker, and J. Denil, "Delfase: A deep learning method for fault space exploration," in *2022 18th European Dependable Computing Conference (EDCC)*, 2022, pp. 57–64.

[19] J. Robbins, K. Krishnan, J. Allspaw, and T. A. Limoncelli, "Resilience engineering: Learning to embrace failure: A discussion with jesse robbins, kripa krishnan, john allspaw, and tom limoncelli," *Queue*, vol. 10, no. 9, pp. 20–28, 2012.

[20] H. Nakama, "Inside azure search: Chaos engineering," Blog, July 2015, microsoft. [Online]. Available: https://azure.microsoft.com/en-us/blog/inside-azure-search-chaos-engineering

[21] P. Joshi, H. S. Gunawi, and K. Sen, "Prefail: A programmable tool for multiple-failure injection," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011, pp. 171–188.

[22] F. Gortazár, M. Gallego, B. García, G. A. Carella, M. Pauls, and I.-D. Gheorghe-Pop, "Elastest—an open source project for testing distributed applications with failure injection," in *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2017, pp. 1–2.

[23] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: A large-scale field study," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 1, pp. 193–204, Jun. 2009.

[24] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 849–867, 2006.

[25] R. Barbosa, F. Cerveira, L. Gonçalo, and H. Madeira, "Emulating representative software vulnerabilities using field data," *Computing*, vol. 101, pp. 119–138, 2019.

[26] F. Cerveira, R. Barbosa, H. Madeira, and F. Araujo, "The effects of soft errors and mitigation strategies for virtualization servers," *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 1065–1081, 2022.

[27] X. Xu and H. H. Huang, "On soft error reliability of virtualization infrastructure," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3727–3739, 2016.

[28] F. Cerveira, J. Domingos, R. Barbosa, and H. Madeira, "Measuring lead times for failure prediction," in *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2021, pp. 1–5.