



UNIVERSIDADE D
COIMBRA

Pedro David Simões de Almeida

**IMPROVING USABILITY AND FUNCTIONALITY IN A FAULT
INJECTION FRAMEWORK**

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor Frederico Cerveira and 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.

September of 2022



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Pedro David Simões de Almeida

Improving usability and functionality in a fault injection framework

Dissertation in the context of the Master in Informatics Engineering,
Specialization in Software Engineering, advised by Prof. Frederico Cerveira and
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.

September 2022

Acknowledgements

First and foremost I am extremely grateful to my parents and sister. Without their tremendous understanding, encouragement and unconditional support in the past few years, it would be impossible for me to complete my study. I would like to express my gratitude to my advisor, Prof. Frederico Cerveira, and co-advisor, Prof. Henrique Madeira, for their invaluable advice, continuous support and providing knowledge and expertise during the project. I am equally grateful to my colleagues and members in this project for their help, feedback sessions and moral support.

The work carried out in this thesis was supported by FCT within project ECSEL/001 8/2019 and the ECSEL Joint Undertaking (JU) under grant agreement no. 876852. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, Turkey.

Abstract

Computer systems are becoming increasingly complex and more prone to faults and failures. Inevitably, failures will occur, and some of these failures can be costly and dangerous. Techniques such as fault injection, either using fault models or failure models, aim to characterise and validate the dependability of a system. Fault injection frameworks have come to be created, however many of them focus on specific types of users, so it is not always easy for a less experienced user to be able to test a system.

One of the objectives of this thesis is the improvement of the usability of ucXception, a framework for performing fault injection in local, virtualized or cloud systems, due to the limitations that it presents. In order to make ucXception more accessible, easier and faster to install and configure, a containerisation technology has been applied. The first phase of the report aims to describe the whole engineering process leading to the realization of the improved version of ucXception, which will be called ucXception 2.0, including a review of the state of the art, a description of the system, i.e. a description of the architecture, technologies used, changes from the previous version, followed by an analysis of the requirements and an overview of the development as well as testing phases.

The final objective achieved was to assess whether fault injection using failure models can accelerate the process of fault injection producing as accurate and representative results. To fulfil this objective, experiments were conducted to compare failure models and fault models with the aim of assessing whether failure models can be used as an alternative to fault models. In order to be able to compare these two models, an existing fault injector tool was used and a new tool was created to inject failures. Openstack, a cloud operating system, was used as the target system for the experiences.

Keywords

Dependability, Fault injection, Fault injection tools, Fault models, Failure models, Usability, Containerization

Resumo

Os sistemas informáticos estão a tornar-se cada vez mais complexos e mais propensos a falhas e avarias. Inevitavelmente, ocorrerão falhas, e algumas destas falhas podem ser dispendiosas e perigosas. Técnicas como a injeção de falha, quer utilizando modelos de falha ou modelos de avaria, visam caracterizar e validar a confiabilidade de um sistema. *Frameworks* de injeção de falhas vieram a ser criadas, no entanto muitas delas concentram-se em tipos específicos de utilizadores, pelo que nem sempre é fácil para um utilizador menos experiente ser capaz de testar um sistema.

Um dos objectivos desta tese é a melhoria da usabilidade da ucXception, uma *framework* que realiza injeção de falhas em sistemas locais, virtualizados ou computação em nuvem, devido às limitações que apresenta. A fim de tornar o ucXception mais acessível, mais fácil e mais rápido de instalar e configurar, foi aplicada uma tecnologia de containerização. A primeira fase do relatório visa descrever todo o processo de engenharia que leva à realização da versão melhorada do ucXception, a ser chamada ucXception 2.0, incluindo uma revisão do estado da arte, uma descrição do sistema, ou seja, uma descrição da arquitectura, tecnologias utilizadas, alterações em relação à versão anterior, seguida de uma análise dos requisitos e uma visão geral das fases de desenvolvimento, bem como das fases de teste.

O último objectivo alcançado foi avaliar se a injeção de falha utilizando modelos de avaria pode acelerar o processo de injeção de falha produzindo resultados tão precisos e representativos. Para cumprir este objectivo, foram realizadas experiências para comparar modelos de avarias e modelos de falhas com o objectivo de avaliar se os modelos de avarias podem ser utilizados como uma alternativa aos modelos de falhas. De modo a conseguir comparar estes dois modelos foi utilizada uma ferramenta que injecta falhas existente e foi criada uma nova ferramenta para injectar avarias. O Openstack, um sistema operativo de nuvem, foi utilizado como sistema alvo para as experiências.

Palavras-Chave

Confiabilidade, Injeção de falhas, Ferramentas de injeção de falhas, Modelos de falhas, Modelos de avarias, Usabilidade, Containerização

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Goals	2
1.3	Document Structure	3
2	Project Management	5
2.1	Semester planning	5
2.1.1	First Semester	5
2.1.2	Second Semester	7
2.2	Process and Organization	8
2.3	Risk Management	10
3	State of the art	13
3.1	Usability	13
3.2	Dependability	14
3.3	Fault Injection	16
3.3.1	Fault Types	17
3.3.2	Techniques for Injection of Hardware Faults	19
3.3.3	Techniques for Injecting Software Faults	21
3.3.4	Fault Injection Tools	23
3.4	Fault injection using failure models	25
3.5	ucXception	26
3.5.1	ucXception architecture	26
3.5.2	Fault injectors of ucXception	29
3.6	Technologies	32
3.6.1	Frontend technologies	32
3.6.2	Backend technologies	34
3.6.3	Docker	34
4	Requirements	37
4.1	User stories	37
4.2	Mockups	38
4.3	Functional requirements	41
4.4	Non-functional requirements	43
4.4.1	Security	43
4.4.2	Usability	44
4.4.3	Extensibility	45
4.4.4	Modifiability	45

5	ucXception 2.0	47
5.1	Architecture	47
5.1.1	Original architecture	47
5.1.2	ucXception 2.0 architecture	48
5.1.3	Choice of technology for each module	50
5.1.4	Entity relationship diagram	51
5.2	Project Structure	52
5.2.1	Frontend module	52
5.2.2	Backend module	53
5.3	Modifications to the original ucXception	54
5.3.1	Campaign configuration file	54
5.3.2	Component configuration file	56
5.4	Containerization	58
5.4.1	ucXception 2.0 containerization	58
5.4.2	Environment setup	60
5.4.3	Extending ucXception 2.0 container	61
5.5	Functionalities	62
5.5.1	Authentication	62
5.5.2	Menu	64
5.5.3	Create campaign	65
5.5.4	Create execution	66
5.5.5	Create host	66
5.5.6	Create component	67
5.5.7	View campaign statistics	69
5.5.8	Build campaign charts	69
6	Testing	71
6.1	Robustness testing	71
6.2	Usability tests	73
6.2.1	Test procedure	74
6.2.2	Test results	75
6.2.3	Test conclusions	79
7	Towards accelerating fault injection using failure models	81
7.1	Methodology	82
7.1.1	Setup	82
7.1.2	Workload	83
7.1.3	Injection process	84
7.1.4	Failure detection	84
7.2	Results	85
7.3	Analysis & Limitations	88
8	Conclusion	89
	Appendix A User stories	99
	Appendix B Mockups	105
	Appendix C Article	125

Acronyms

AC Acceptance Criteria.

ALU Arithmetic Logic Unit.

API Application Programming Interface.

ARM Acorn RISC Machine.

AST Abstract Syntax Tree.

CPU Central Processing Unit.

CSV Comma-Separated Values.

ER Entity Relationship.

FI Fault Injection.

FPGAs Field Programmable Gate Arrays.

FPU Float Point Unit.

HTML HyperText Markup Language.

HTTP Hyper Text Transfer Protocol.

HWIFI Hardware-implemented fault injection.

I/O Input/Output.

IP Internet Protocol.

ISA instruction set architecture.

PID Process Identifier.

PRDC IEEE Pacific Rim International Symposium on Dependable Computing.

REST Representational State Transfer.

SCIFI Scan Chain Implemented Fault Injection.

SQL Structured Query Language.

SSE Software and Systems Engineering.

SSE Streaming SIMD Extensions.

SSH Secure Shell.

SW Software.

SWIFI Software-Implemented Fault Injection.

URL Uniform Resource Locator.

US User Story.

VM Virtual Machine.

XML Extensible Markup Language.

List of Figures

2.1	Plan for the first semester.	6
2.2	Actual timeline for the first semester.	6
2.3	Plan for the second semester.	8
2.4	Actual timeline for the second semester.	9
3.1	The fundamental chain of dependability threats.	15
3.2	Diagram of software faults types [23].	19
3.3	ucXception architecture [11].	27
3.4	Hardware fault injection flow [11].	30
3.5	Fault injection flow for virtualized systems [11].	30
3.6	Software fault injection flow [11].	32
3.7	Docker architecture diagram [17].	35
4.1	Login page mockup.	39
4.2	Menu page mockup.	40
4.3	Campaign configuration page mockup.	40
4.4	Campaign menu page mockup.	41
5.1	ucXception original architecture.	48
5.2	ucXception 2.0 architecture.	49
5.3	Conceptual diagram.	51
5.4	Login, register and password reset forms.	64
5.5	Menu page.	65
5.6	Create campaign page.	65
5.7	Create execution page.	66
5.8	Create host page.	67
5.9	Create component page.	68
5.10	Page to create a transformer that is not valid.	68
5.11	Campaign statistics page.	69
5.12	Campaign charts page.	70
6.1	First question.	77
6.2	Second question.	77
6.3	Third question.	77
6.4	Fourth question.	78
6.5	Fifth question.	78
7.1	Experimental Setup.	83
7.2	Failed operations per run for both models.	85
7.3	Failures per operation for both models.	86

7.4	Distance between failure model and oracle.	87
7.5	Distance between failure model and oracle.	87
B.1	Login page.	106
B.2	Registration page.	107
B.3	Password reset page.	108
B.4	Password reset confirmation page.	109
B.5	Menu page.	110
B.6	Campaign statistics page.	111
B.7	Campaign graphics page.	112
B.8	Campaign raw data page.	113
B.9	Campaign configuration page.	114
B.10	Campaign pre-probes configuration page.	115
B.11	Campaign post-probes configuration page.	116
B.12	Campaign parsers configuration page.	117
B.13	Campaign validators configuration page.	118
B.14	Campaign transformers configuration page.	119
B.15	Campaign configuration summary page.	120
B.16	Campaign summary configuration page.	121
B.17	Campaign pre-probes configuration summary page.	122
B.18	Campaign post-probes configuration summary page.	123
B.19	Campaign parsers configuration summary page.	124

List of Tables

2.1	Risks.	11
3.1	Fault Injection Tools.	24
3.2	Differences between fault models and failure models.	25
3.3	Pre-defined components.	28
3.4	Software faults operators [11].	31
3.5	Comparison of frontend technologies.	33
3.6	Comparison of backend technologies.	35
4.1	User feedback.	39
4.2	Functional requirements of ucXception 2.0.	42
5.1	Implemented ucXception 2.0 requirements.	63
6.1	Experimental setup specification for robustness testing.	72
6.2	Example of cases where the API has failed.	72
6.3	List of problems encountered for each case.	73
6.4	Time taken for each task performed by the participants.	75
6.5	Number of clicks on each task performed by the participants.	76
7.1	Experimental setup specification	82

Chapter 1

Introduction

One of the big problems with the evolution of systems both at hardware and software level is related to ensuring that the system does not fail and does not provide a wrong service that can negatively affect those who use it. In order to solve this problem one of many solutions is adopted, such as fault injection techniques to catalyse the process of fault activation in a target system [57]. There are a variety of fault injection techniques to help evaluate the dependability of a system, from techniques to inject hardware faults to software faults [6].

Consequently, injecting a fault in a system requires a prior study of what faults one wishes to emulate as many different faults, such as hardware, software or operator faults, can affect a system, and each fault must be emulated in a specific manner. To this end, concepts arise, such as: fault types, which define the type of system fault (i.e. hardware or software faults) and are classified according to their appearance and duration, and fault models that define an unusual behaviour for which the system may not be prepared [44].

This thesis is integrated in the European project VALU3S¹ which evaluates the state-of-the-art verification and validation (V&V) methods and tools for automated systems in the automotive industry, agriculture, railway, healthcare, aerospace and industrial automation and robotics.

The thesis focuses on improving ucXception [11, 50], a framework that allows the injection of hardware or software faults, through several tools, in a local or remote system. The framework was developed by people from the Software and Systems Engineering (SSE) group in order to be able to study the area of fault injection. ucXception provides several components that allow data collection for a more complete and simple analysis. The fault injection tools allow faults to be injected either in Linux or virtualized systems and the possibility of adding new tools is verified by the way the framework was developed.

Nowadays, fault injection using fault models is the most approached concept when we want to evaluate systems, however fault injection using failure models is another possible technique for this purpose. Failure models, which aims to directly emulate a system failure instead of introducing the fault that causes the

¹VALU3S Page <https://valu3s.eu/>.

system to fail, has not yet been thoroughly discussed, but promises faster assessment than fault models. When approaching fault injection using failure models [28, 40] we must apply the same study procedure of fault models, that is, define what is really intended to be studied and injected into a system. Failure models aims at a faster assessment and as representative as fault injection.

1.1 Motivation

The ucXception framework offers several advantages, allowing hardware and software fault injection in distributed systems and supporting fault injection in virtualized systems, however it contains limitations. Two main reasons that led to the execution of this thesis come from the problem pointed out to ucXception. The first reason is because the tool has bad usability, does not have an user interface, therefore is hard to use and the the second reason comes from limitations in setting up the framework, which is more time-consuming and requires more effort.

The lack of an evaluation of the representativeness of failure models and a comparison between fault injection using failure models and fault models is the motivation behind the third objective. Comparing results between the two techniques, fault and failure models, will allow to draw clearer conclusions about the use of failure models. Aiming to investigate if failure models is a good approach to accelerate campaigns in a representative way is the second motivation of this thesis.

The motivations pointed out led to the creation of essential objectives for the development of this work, which will improve the weaknesses of the ucXception framework and make it a more useful tool for different types of users who want to test their systems, and will allow a comparison between two currently fundamental concepts, fault and failure models.

1.2 Goals

In total there are two main objectives to be accomplished and completed that stem from the problems of the ucXception framework pointed out in Section 1.1 and from a research component.

As one of the main motivations is the low usability and difficult configuration of the framework, the first goal is to improve usability so that a wider audience can use it even if they are not experts in the field and make the framework more accessible, i.e. encapsulate the application in a single compartment and therefore make it quick and easy to configure in a system. This step initially involves collecting requirements, design user interfaces, planning an architecture for the new version of the framework, ucXception 2.0, studying technologies for development and packaging as well as testing the framework for both implementation bugs and usability.

The second objective is to accelerate the fault injection process. For this purpose, a new failure model tool will be developed, experiments will be conducted with failure and fault models and the results will be analysed to draw a conclusion. This step consists in evaluating if using failure models are as accurate and representative as fault models, if the hypothesis is true it will lead to faster campaigns and a tool should be implemented in the framework in order to cover more analysis methods for the users.

It is aimed will be to write an article on the new version of the framework developed, presenting its usefulness and advantages as well as explaining its architecture and finally presenting the research process carried out in respect of the comparison between failure and fault models.

1.3 Document Structure

This preliminary report is divided in 8 chapters: Introduction, Project Management, State of the art, Requirements, ucXception 2.0, Testing, Towards accelerating fault injection using failure models and Conclusion.

The first chapter identifies the context of this report, the motivation, the main objectives and presents the structure of the document.

The second chapter presents the temporal planning of the work done, the process and organisation during the development and finally the risks considered in this project.

The third chapter (State of the Art) presents the study that was done to deepen the knowledge of the themes of this dissertation. First a brief study is made regarding usability and its importance for a good functioning and user adherence. Secondly, dependability is defined, an important topic when systems are to be evaluated. Following, a technique to assess dependability, fault injection, is presented, addressing some of fault types that can be injected and their classification and a clarification about fault models. Some fault injection techniques are presented for both hardware and software and a list of tools that implemented these techniques. Furthermore, the topic of fault injection using failure models is presented explaining its purpose and adversities. The next section presents the ucXception tool starting by identifying the reasons and objectives for which it was created. Its architecture is explained and the fault injection flows of the tools that constitute it. The last section addresses the technologies studied for the development of the framework.

The fourth chapter presents all the techniques of requirements gathering and the respective requirements gathered, such as user stories, mockups, functional and non-functional requirements.

The fifth chapter start with an explanation of the original architecture and the architecture planned for ucXception 2.0, then the technologies that were taken into consideration for each module of the architecture and a detailed explanation of the relational diagram built. Furthermore, it presents the project structure for

both implemented modules, frontend and backend, the modifications made to the original version of ucXception, the application of containerization technology and finally a detailed approach to all the functionalities present in ucXception 2.0.

The sixth chapter (Testing) addresses the two types of tests performed to verify and validate the new version of the framework both at the implementation and usability levels.

The seventh chapter presents the methodology followed, such as setup, injection process among others, an analysis of the experiment results and finally a brief discussion of the limitations of the experiment conducted.

The last chapter summarizes the report and reveals the final ideas for the project and future work.

Chapter 2

Project Management

This chapter focuses on the main aspects relevant to project management. It starts by explaining the planning done for each semester and a brief discussion on the deviation of the planned plan with the actual timeline. Then the process and organisation followed during the development of the framework. At the end, there is a presentation of the risks that were considered in the project.

2.1 Semester planning

The weekly meetings held served to check the progress in a more continuous way, which allowed the work not to accumulate. The meetings essentially helped to clarify some of the doubts that arose, both in terms of the state of the art, the requirements and some development aspects.

In order to manage and control the schedule of activities required for each semester a visual tool such as the Gantt chart was used. Furthermore, in order to have a comparative term between the planning and the actual course of the semesters, at the end of each semester, the deviation between the two was analysed in order to understand what went more negatively, serving as learning for future projects.

2.1.1 First Semester

The first semester was used to study the state of the art, to understand the concept of the framework, to do requirements gathering and to plan in detail the development process that took place in the second semester. Initially a Gantt chart was created (Figure 2.1) to plan the course of the first semester. A second chart (Figure 2.2) was made in order to show the actual timeline during the first semester. The Gantt chart has 6 stages:

- **Study the state of the art:** Study thesis related area such as dependability, fault injection using fault as well as failure models, technologies among other topics;

- **Explore ucXception:** Explore the ucXception framework in order to understand its main functionalities;
- **Write user stories:** Write user stories in order to gather requirements;
- **Design mockups:** Design mockups for the graphical interface;
- **Plan architecture:** Plan an architecture to be developed in the second semester;
- **Write intermediate report:** Write the preliminary report in order to present the work already done.

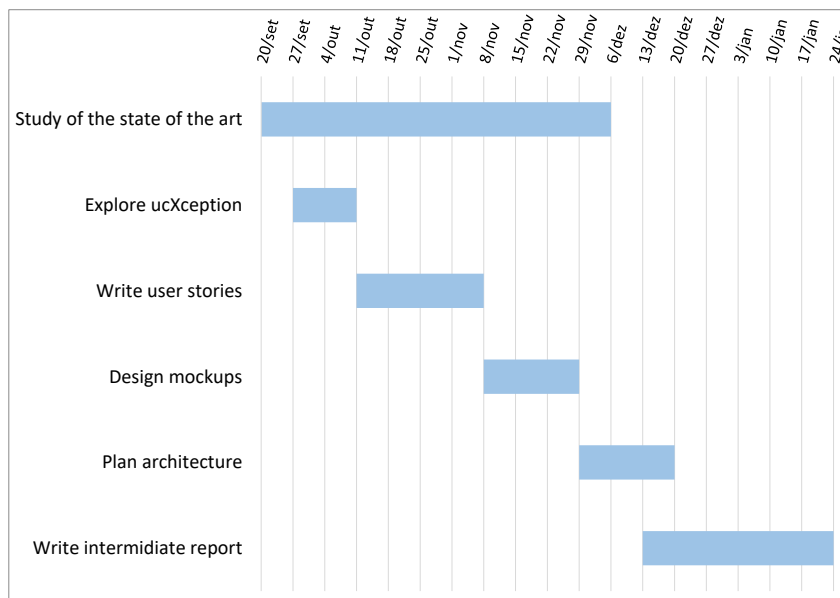


Figure 2.1: Plan for the first semester.

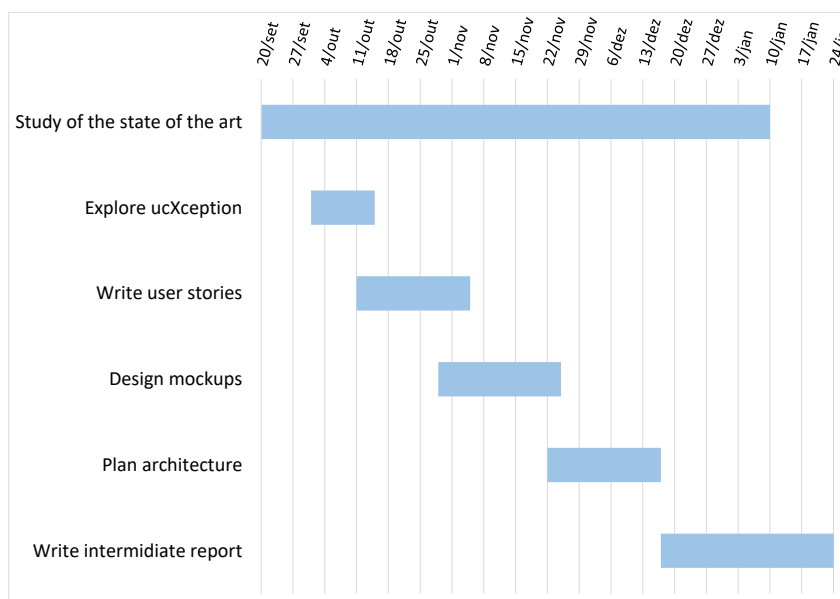


Figure 2.2: Actual timeline for the first semester.

It is concluded from the analysis of the charts that there have been changes and delays with regard to the plan designed at the beginning of the semester.

Firstly, the difference that stands out the most is the state of the art study stage which lasted about 5 weeks, this is due to the fact that during the writing of the preliminary report there were always topics that were important to mention, but which had not yet been studied.

Exploring the framework was slightly delayed due to the delay in the state-of-the-art study. The following stages were a bit off schedule, as the requirements gathering through the user stories was progressing it was possible to anticipate the other stages. The reasons why the mockup design and architecture planning were delayed were due to several essential factors that came up during the meeting reviews. Consequently, the writing of the report was slightly delayed.

2.1.2 Second Semester

The second semester was planned according to the objectives to be achieved and a most likely chronology is considered, i.e. neither best case nor worst case (see Figure 2.3). The Gantt chart has 5 stages:

- **Implement and test of GUI of ucXception 2.0:** Develop the graphical interface;
- **Containerize ucXception 2.0:** Containerise the framework;
- **Evaluate failure models for accelerate fault injection:** Develop fault injection tool, carry out campaigns, analyse obtained results;
- **Write research paper:** Write a research work related to the new version of the framework and to the experiments performed to compare fault and failure models;
- **Write final report:** Write the final report in order to present the work done.

Initially, the plan starts with the development of the graphical interface and after some progress, having the structure of the framework already built, start applying containerisation. The development and execution of fault injection through failure model campaigns is planned to be done in parallel during the development of the framework in order to optimise the time available. The two final objectives are to write a paper related to the new version of the framework and the conducted fault injection experiments using fault models as well as to write the final report.

The actual timeline of the second semester had some deviations compared from what was planned. Starting with the implementation of ucXception 2.0, it was delayed due to the intermediate defense held on the 7th of February and because after the defense it was decided to focus on the comments received regarding the first phase. In addition, lack of experience with some technologies and adaptation

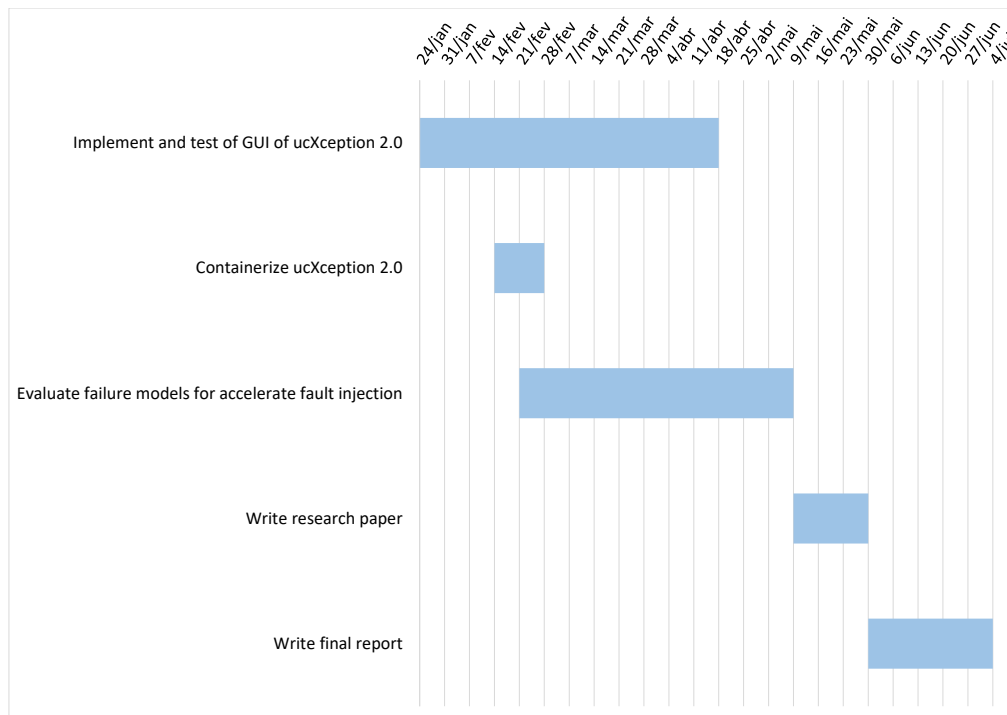


Figure 2.3: Plan for the second semester.

of the framework made it more delayed, so it was identified as a risk. The time spent on average on each functional requirement is presented, highlighting more the delay in the creation of components and in the part of analysing the results.

With the delay in implementation, containerisation was postponed but remained on schedule for approximately two weeks. In the second semester planning the validation of the framework was not addressed, however two types of tests were performed which took about 3 weeks, with usability tests taking up more time.

Regarding the experiences, this was the biggest deviation from what was expected, due to the delay in the preparation of the experimental setup and the time the experiences took to execute. Furthermore, there were experiences that failed in which several results were lost and the experiences had to be re-run. As a consequence of these delays the writing of the paper and the final report were also postponed, although the time allotted for writing the paper was met and the time taken to write the final report was longer than anticipated.

2.2 Process and Organization

In all software projects it is crucial to have a good organization especially when we talk about development tasks that can cause delays in the project or even cause lack of functionality in the early release project. Thus, the organisation of tasks, the technologies adopted for the organisation of code or responsibilities are elements to be taken into consideration before starting the development of the project.

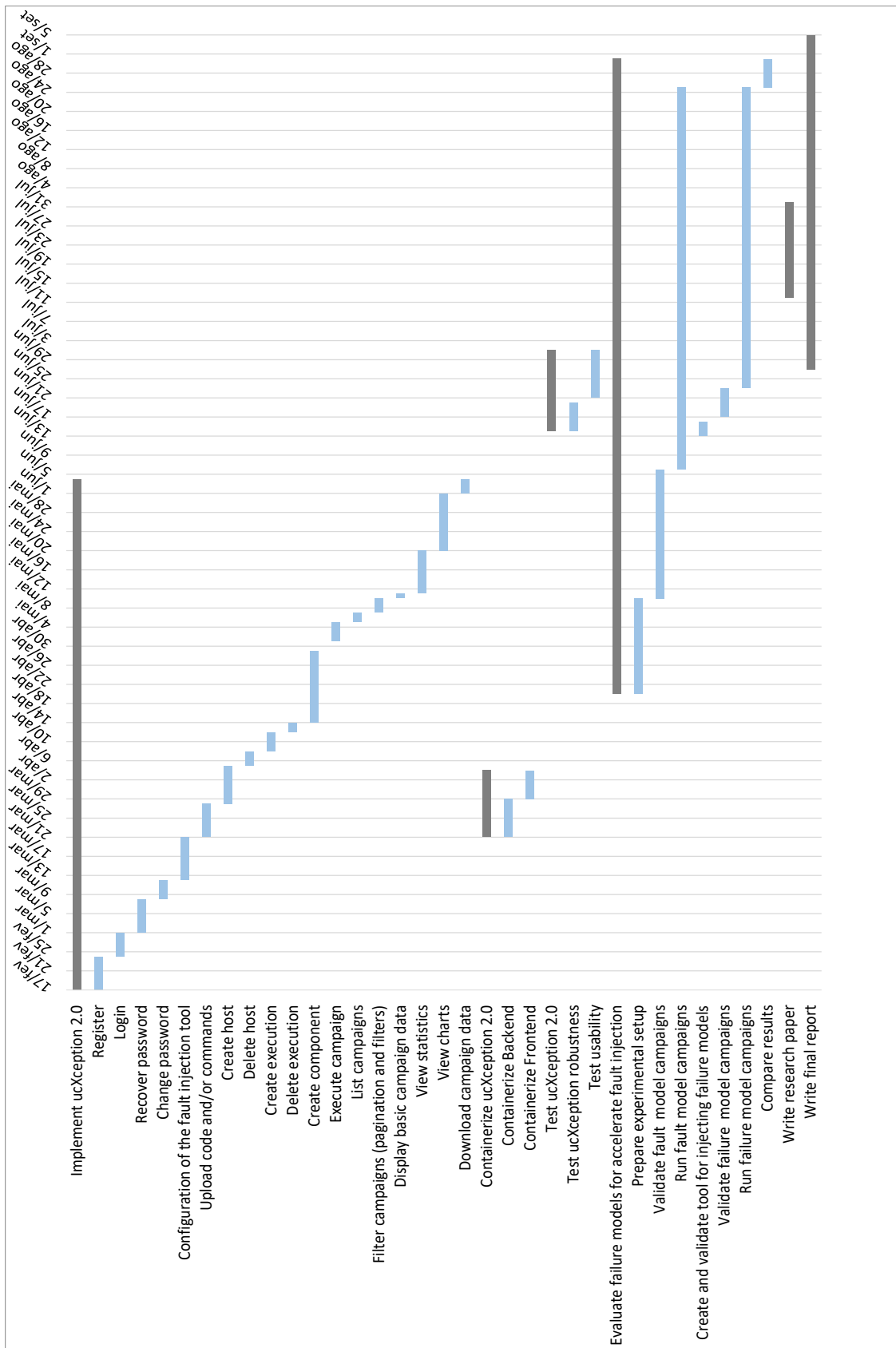


Figure 2.4: Actual timeline for the second semester.

Regarding the organisation of the tasks, a list was made based on the functional requirements and the objectives of the thesis with the various functionalities to implement. The list served essentially to track the tasks and to avoid leaving any functionality unimplemented. Each task required development for both the backend and the frontend starting with developing the backend module. The process involved for each task developing, testing and, if no errors were found, proceeding to the next task. In order to verify and validate that the development was meeting the requirements and the client's expectations, weekly meetings were held to demonstrate the project status and even to clarify aspects that were less explicit.

Concerning the technologies, essentially two were used, Github and Docker HUB. The use of Github, a platform for hosting source code and files with system version control, allows the project to be kept in a central repository and control over the completed tasks. The repository ¹ has a default branch, *master*, which in the case of this project contains the original code of the ucXception framework. A branch allows to change parts of resources in the repository without affecting other branches. In order to create the new version of the framework, a new branch called *dev* was created. The branch *dev* contains two folders one for the development done for the frontend module and one for the backend module. After the conclusion of each task, a new version of the system was created in the *dev* branch, obtaining a history of the versions in the repository.

The Docker HUB is a technology that allows sharing a specific repository of the technology so that users have access to the new version of the framework faster and more simplified. In addition, the use of this repository ² eases the process of testing with users regarding the configuration of the framework.

2.3 Risk Management

An approach to risk management involves the identification, analysis/assessment, mitigation and monitoring of risks during the life of a project to minimise the chances and severity of an event occurring. The identification of risks is the process of determining and assessing potential threats to a project. Once the risks have been identified, the objective of risk analysis is to identify the probability of a risk event occurring and the potential outcome. The severity of each risk is compared and ranked according to its prominence and consequences. In the final step of the project management process, risk mitigation involves devising and implementing methods and options to lower threats against the project's objectives. Implementing risk mitigation strategies can help identify, monitor, and assess risks and consequences inherent in the completion of a specific project. The following will show the risks identified where they can be classified as:

- **Probability of occurrence**

¹ucXception GitHub <https://github.com/ucx-code/ucXception/>.

²ucXception image <https://hub.docker.com/r/pedroalmeida705/ucxception>.

- **Low:** The risk has a lower than 40% probability of occurring;
- **Medium:** The risk has a 40% to 70% probability of occurring;
- **High:** The risk has a higher than 70% probability of occurring;

- **Impact for the realisation of the project**

- **Low:** No significant deviations in terms of effort and time consumption and no significant impact on scope;
- **Medium:** Effort and calendaring should be readjusted within the allotted time and scope will be met by reallocating resources;
- **High:** Immediate adjustments are needed with additional effort than planned.

The following is a list with the identified risks and Table 2.1 with the analysis of the risks in relation to the probability of occurrence and the impact it has on the project. It also contains the respective mitigation plan and a column indicating whether or not it happened.

- **Risk1** - During the execution of experiments some error may occur (lack of computational resources, programming error, etc...).
- **Risk2** - During the adaptation of the tool to a new Python version and generalising various aspects can take more time than expected.
- **Risk3** - Due to the lack of experience, the expected time for the execution of each objective presented in the Gantt may have delays in the final project.

ID	Probability	Impact	Mitigation Plan	Occurred
Risk1	Medium	High	Analyse the problem, solve it and re-run the campaigns.	Yes
Risk2	High	High	More detailed analysis of the topic. In addition, the advisor, being familiar with the framework, can help with any complex issues.	Yes
Risk3	Medium	Medium	Meetings with the advisor, in order to validate the planning and validate which requirements may not be so important.	Yes

Table 2.1: Risks.

Regarding the first risk, during the execution of the experiments several errors occurred in which it was necessary to apply the mitigation plan to solve the errors. These errors caused a delay in the execution and analysis of the experiments, as it was expected to obtain an accessible number of results, which was not possible.

The second risk occurred essentially in the generalization of components and campaigns, one of the most important tasks in the construction of the new version of the framework that allows users to add their own elements in the framework. For the campaigns the delay was slight, however each component was built differently so the delay was longer. This risk was expected to occur, but by analysing possible solutions with the supervisor it was mitigated.

The last risk also occurred but it had the least impact and least likelihood of occurring in several objectives, so the mitigation plan worked perfectly. One example was the final summary after creating a campaign and the components that were left for future development.

Chapter 3

State of the art

This chapter explores all the topics that are associated with the scope of the ucXception framework, thus creating a knowledge base that made it possible to understand all the context surrounding the theme of the project. The chapter is divided into five sections, which are:

Section 3.1 presents a study regarding usability based on several attributes and how to test the usability by validating it in the best possible way.

Section 3.2 explores the concept of dependability and the importance that this attribute has for today's systems. It begins with a brief introduction on the concepts related to dependability and leads into the definition of dependability.

Section 3.3 addresses the topic of fault injection by presenting a definition, its purpose, some important concepts for fault characterization and some fault injection techniques, as well as several tools that implement them.

Section 3.4 addresses the topic of fault injection using failure models presenting a general concept and comparing it with fault models.

Section 3.5 provides a brief introduction to the ucXception framework, what it is for and what the goals are. The ucXception architecture is presented, addressing its components, fault injectors delivered by the framework and the respective injection flows.

3.1 Usability

In order to understand the subject of usability there are 3 questions that are most commonly addressed: what is usability, why is usability important, and how to improve usability [46]. To answer the first question, usability is defined as a quality attribute that evaluates how easy is to use a user interface and whether a product can be used by specific users to accomplish certain goals effectively, efficiently and to their satisfaction [1]. There are five quality components that define usability [46]:

- **Learnability:** “How easy is it for users to accomplish basic tasks the first time they encounter the design?”
- **Efficiency:** “Once users have learned the design, how quickly can they perform tasks?”
- **Memorability:** “When users return to the design after a period of not using it, how easily can they reestablish proficiency?”
- **Errors:** “How many errors do users make, how severe are these errors, and how easily can they recover from the errors?”
- **Satisfaction:** “How pleasant is it to use the design?”

The attributes presented are among the most important that should be taken into consideration when addressing usability, however there are many other important quality attributes, such as utility. Utility is defined as a product that provides the features a user needs. Usability and utility work together, as a product may provide the user with what is needed but it is complex to perform the task through the graphical interface or vice versa. The second question can then be answered, usability is important as it is a matter of the user choosing the simplest product to use and one that meets their requirements.

The third question is related to the assurance that the attributes are fulfilled. For this, the method used is usability testing usually done on a stable version of the product, which requires representative users. During a usability test, participants typically attempt to complete tasks individually, letting them solve any problems on their own while observers watch, listen, and take notes. Identifying usability issues, collecting qualitative and quantitative data, and assessing participant satisfaction are the main objectives [65]. The problems encountered lead to changes in the application making the process of usability testing important before the launch of the product to ensure that it is suitable for deployment.

Usability testing requires prior preparation, i.e. planning the tasks and the flow, the metrics to be collected during testing and post-testing and choosing participants with product user characteristics that match the target audience to provide the most accurate results possible [62]. Test plans can be designed once all the necessary information has been gathered.

3.2 Dependability

When discussing dependability it is important to know basic concepts beforehand, such as fault, error and failure and the impact that these threats have on a system.

A service provided by a system, as a provider, is the behaviour as perceived by another system receiving service. A correct service is a service when it performs as intended, on the other hand, service failure, or failure, is an event that occurs

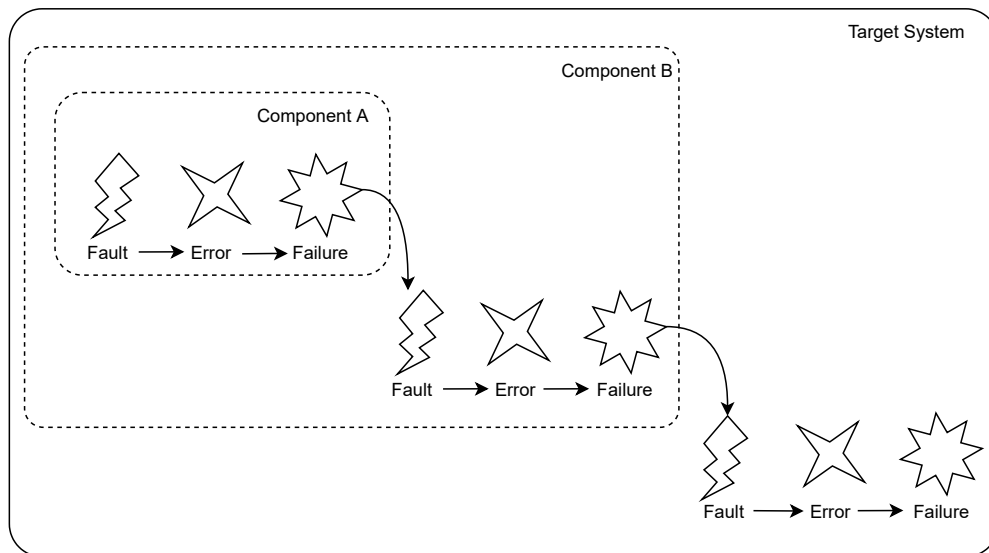


Figure 3.1: The fundamental chain of dependability threats.

when it deviates from the expected behaviour. This occurrence stems from non-compliance with the specifications of a service. When an incorrect system state occurs, referred to as an error, the system deviates from the correct service state. The error is hypothetically caused by a fault [5], as depicted in Figure 3.1.

Faults can be internal or external to the system and when they occur they can trigger one or more latent errors which when activated can be detrimental to the system. Latent error is defined as “errors that are present but not detected” [5]. An active error can create more errors, which in turn can affect the service, thus generating a failure. The system failure will not occur until the error reaches the boundaries of the provider’s system where the service delivery takes place.

Depending on the service these failures can cause problems ranging from security threats to service unavailability and incorrect output being produced, stored or transmitted to outside of the system. These problems lead us to define dependability.

Avizienis et al. [5] present two definitions of dependability:

- **Original definition:** “dependability is the ability to deliver service that can justifiably be trusted”;
- **Alternate definition:** “dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable”.

The original definition needs a definition of trust. Dependency between systems can be total or none. For instance, a system A that is totally dependent on a system B, can be affected by a failure of system B. Whereas not being dependent the probability of causing a failure is zero. This concept of dependency leads to the concept of trust that needs to be accepted between these two systems.

The best way to understand dependability is to understand the attributes that it encompasses. Avizienis et al. [5] presented the following attributes:

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

Consequently Avizienis et al. [5] identified a group of four means to reach dependability:

- **Fault prevention:** Means to avoid occurrence of faults. We can prevent development faults using development methodologies for software and hardware;
- **Fault tolerance:** Means to prevent service failures when faults exist in the system;
- **Fault removal:** Means to decrease severity and the number of faults during development and during system use. During development phase of a system consist in three steps: verification, diagnosis and correction. During use of a system consist in corrective or preventive maintenance.
- **Fault forecasting:** Means to estimate the fault occurrence and activation by performing an evaluation of the system behaviour. Considering qualitative and quantitative evaluation.

In order to make sure that the system can be “trusted” a technique for evaluating dependability in systems is presented in Section 3.3.

3.3 Fault Injection

Over several years, with the increasing complexity and demands on systems, the tendency to detect hardware faults and software faults, or so called software defects or bugs, and validate fault-handling mechanisms is becoming more and more usual and necessary. A system that provides a service with failures more frequent and more severe than acceptable cannot be trusted, thus its dependability is poor [5].

In order to assess and verify the fault-handling mechanisms and evaluate dependability measures a practical approach can be used, fault injection [6]. This experimental technique deliberately introduces faults in a system [44]. Usually faults can be hardware or software [6] and are injected in a real system, in a physical computer system, or in a system model that accurately reflects the behaviour of the system components.

It is relevant to have a notion of the importance of fault models when we talk about fault injection. Fault models characterize the faults that a system will be subject to during operation and must follow a key property: representativeness. This property can be defined as “the ability of the faultload and of the workload to represent the real faults and inputs that the system will experience during operation” [44]. It can be achieved by defining a realistic fault model following three important aspects: the type of faults to inject (“What to inject”), fault locations, system component targeted by the injection (“Where to inject”) and the injection timing (“When to inject”). We can have several models modelling faults for the same purpose, however depending on the choices made in response to these questions, some will be more realistic than others. This is specially true for the failures models often used by fault injection, which tend to lack precision when compared with most fault models used for fault injection. The topic of fault injection using failure models will be further developed in Section 3.4

Fault injection involves concepts such as fault injection experiment and fault injection campaign. The fault injection experiment represents the injection of a fault and the observation of the system behaviour. A set of fault injection experiments defines a fault injection campaign. In order to evaluate the results of fault injection experiments, experiments without fault injection are used as a point of comparison, so called golden runs [44].

There are several fault injection techniques that will be explained in more detail in Section 3.3.2 and Section 3.3.3. Every technique can be classified by the following properties:

- **Controllability:** Ability to control fault injection, both in time and space;
- **Observability:** Ability for observing and recording the consequences of a fault injection;
- **Repeatability:** Capability of repeating a fault injection experiment and obtaining the same result;
- **Reproducibility:** Be able to replicate the results of a fault injection campaign;
- **Representativeness:** The degree of accuracy that the fault injection experiment actually represents, the real system and the injected faults.

3.3.1 Fault Types

Raul Barbosa et al. [6] presented two main fault types that can be injected: hardware faults and software design and implementation faults. The most common fault models for emulating hardware faults are stuck-at-one, stuck-at-zero or bit-flip affecting computer’s components such as Central Processing Unit (CPU) registers, main memory or communication buses [57]. The most common fault models for emulating software faults are based in two principles: altering source code by replicating programmer errors or error injection. Error injection attempts to

emulate software faults by changing values of system state variables such as CPU registers and data store or changing parameters of code functions.

Hardware fault types can be classified as:

- **Intermittent:** appears and disappears several times in a seemingly random and unpredictable [5];
- **Permanent:** remain active until the system is repaired [5];
- **Transient:** temporary in nature as it appears and eventually goes away [5]. Transient faults are caused by environmental effects, such as cosmic rays, usually referred to as soft errors [43].

Software fault can be classified into the following types:

Bohrbugs are faults that are easily isolated and which manifests reliably under a well-defined, but possibly unknown, set of conditions. These faults are easy to detect and reproduce because of their lack of complexity [23].

On the other hand, one can define **Mandelbugs** as the complementary antonym of Bohrbugs. In other words, a complex type of bug that is difficult to fix because of its complexity and unpredictability. Under apparently identical conditions, sometimes failure occurs, while on other occasions no failure is experienced, not reproducible [23].

Due to the subjectivity in distinguishing the complexity of the circumstances that led to a failure between these two concepts, the classification can be more objective by presenting two cases that consider an application failure as a Mandelbug [23]:

- If elements of the software system other than the application itself influence the cause of a failure;
- If the complexity of error propagation results in a delay between the fault activation and the behaviour perceptible by another system.

The two cases divide Mandelbugs into two sub-types. Regarding the first case, it is mentioned as **Heisenbug**, a software bug that, after study/observation through a tool or method, can disappear or manifest differently. The second case is referred as **Aging-related bugs**, occur in long-running systems due to error conditions caused by the accumulation of problems such as memory leaks or round-off errors in program variables, “resulting in an increased failure rate and/or degraded performance” [23]. These two classes, Heisenbug and Aging-related bugs, may or may not overlap with respect to a specific observation tool or method.

Figure 3.2 shows the relationship of software fault types in a diagram.

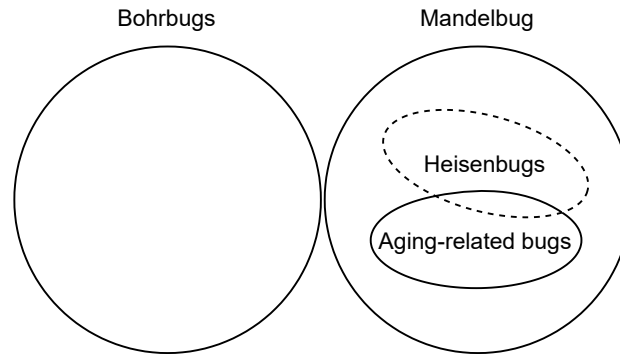


Figure 3.2: Diagram of software faults types [23].

3.3.2 Techniques for Injection of Hardware Faults

As mentioned in Section 3.3 fault injection can be executed in real systems or in a model of a system. Fault injection in these types of systems has advantages and limitations.

Injecting faults into real systems brings a more effective evaluation and verification of dependability, its properties and fault tolerance mechanisms. On the other hand, simulation-based and emulation-based fault injection recreate realistic faults more accurately than faults injected into real systems. However simulation and emulation models have their limitations, such as development cost, reproducibility, and time-consuming, it may not be practicable to use a detailed model with a large amount of activity.

Some properties such as controllability, observability, repeatability and reproducibility are normally higher when injecting in simulation and emulation models than injecting in real systems [6, 44].

There are three commonly addressed fault injection techniques used in real systems [6]: Hardware-implemented fault injection (HWIFI), Software-Implemented Fault Injection (SWIFI), and Radiation-based fault injection, and two techniques for model systems, Simulation-based and Hardware emulation-based fault injection.

In the following subsections a general concept of each method is provided.

Hardware Implemented Fault Injection

In this subsection three techniques for fault injection are introduced: pin-level, power supply disturbances, and test port-based.

Pin-level method inject faults via probes attached to the pins. It is limited to stuck-at faults. Essentially this technique is used in automotive and industrial embedded systems [6, 33, 70].

Power supply disturbance faults are injected by attaching active probes to the power supply. This technique is not commonly used due to its low repeatability

and tends to affect many bits, impossible to emulate faults where only one bit is flipped, which can damage the injected device [25].

Test port-based uses built-in debugging and testing features from microprocessors to inject faults. This type of faults can be injected in all registers of the instruction set architecture (ISA). ISA is part of the abstract model of a computer that defines how the CPU is controlled by the software. The time it takes to inject the faults depends on the speed of the test port. The main advantage of this method is that it is not necessary to make any changes to the hardware components [25].

Software-Implemented Fault Injection of Hardware Faults

SWIFI of Hardware Faults is a technique that allows injecting hardware faults through software. Several problems associated with physical fault injection techniques (e.g., pin-level injection and heavy-ion radiation) were overcome by SWIFI, such as repeatability, controllability and simplicity [44].

This method has advantages such as:

- It can be used to target applications and operating systems, which is difficult to do with hardware fault injection [59, 70];
- It does not require an specific hardware and cannot damage the target system [59, 70];
- “It has high controllability over where and when faults are injected and high observability of fault behavior and fault propagation” [59].

As well as disadvantages:

- Impossible to inject faults into locations that are inaccessible to software [59, 70];
- “The software instrumentation may disturb the workload running on the target system and even change the structure of original software” [59];
- It is very difficult to model permanent faults, because requires a more elaborate set of manipulations [6, 70].

There are two distinct ways to inject faults using this method: run-time and pre run-time injection. In run-time injection, faults are injected while a workload is being executed by the system. This approach implies a significant runtime overhead. In pre run-time injection, faults are insert by manipulating the source code and the binary image of the workload, before it is executed. Compared to run-time injection, pre run-time incurs less runtime. However by taking longer to prepare each fault injection experiment the total runtime of the fault injection campaign is longer.

The beginning of research in this area has led many authors to create their own fault injection techniques supporting different fault models. For example, FIAT [54], FERRARI [30], FINE [31], FTAPE [64] and Xception [10]. In Section 3.3.4 the different models used by each tool are presented.

Radiation-Based Fault Injection

Various systems and electronic integrated circuits are sensitive to external disturbances like electromagnetic interference and particle radiation.

As explained in Section 3.3.1, this type of disturbance can cause errors that appear and eventually disappear, so called soft error. This technique provides a validation method by exposing the system, sensitive regions within a circuit, to ionising particles.

Raul Barbosa et al. [6] concluded that this technique “has very low, or non-existent, repeatability” and it is very difficult to control the time and place of the injection due to the lack of precision of controlling heavy-ion emission. In this way, it is not possible to reproduce an experience.

Simulation-Based Fault Injection

Simulation-Based Fault Injection uses a software program to simulate hardware faults on a model of a system, generally called fault simulator. Fault injection can occur during run-time, injected during the simulation run or compile-time, and are injected into the target hardware model.

The advantages of this technique are that there is no risk to damage the system, it is less costly in terms of time and effort and has a higher controllability and observability of the system behavior in presence of faults. However, it lacks accuracy in the system model and fault model [33].

Hardware Emulation-Based Fault Injection

This technique has emerged in order to reduce the time that a fault injection experiment takes compared to simulation-based fault injection. To achieve that, hardware emulation-based fault injection is based on the use of Field Programmable Gate Arrays (FPGAs), an integrated circuit designed to be programmed by the customer or a designer after it is manufactured [6, 33].

3.3.3 Techniques for Injecting Software Faults

In addition to hardware fault injection, another important topic in the field of fault injection study is software fault injection or software fault emulation.

Software faults are one of many causes of system outages. Given the enormous

complexity of the current software, the software faults tend to increase [20, 48, 68], so it is relevant to extend fault injection technologies to the injection of this type of faults.

To date, effort has been invested to create techniques that can tolerate and handle software faults. In order to validate these techniques, fault injection techniques, which can accurately mimic the impact of real software faults, have been developed. It has been recognized that fault injection can be used to emulate the effects of real software faults [12, 66], allowing the assessment of hidden bugs in a system through accurate ways of emulating software faults.

Raul Barbosa et al. [6] presented two fundamental approaches to software fault injection: fault injection and error injection. Fault injection imitates programmers' errors by altering the code executed by the target system, while error injection attempts to imitate the consequences of software failures by manipulating the state of the target system.

Emulating Software Faults by Error Injection

Software-implemented fault injection for hardware faults drove studies based on software fault injection approaches. Mainly SWIFI allowed to produce errors by injecting hardware faults (Section 3.3.2). Later on SWIFI approaches were adopted to emulate software faults by error injection [7].

Raul Barbosa et al. [6] refer two common techniques:

- **Program state manipulation** involving change of variables, pointers and other data stored in main memory or CPU registers;
- **Parameter corruption** corresponding to the modification of parameters of functions, procedures and system calls.

The main challenge is to find error sets that are representative of real software faults in which each error is defined by: error type (what to inject), error location (where to inject), injection time (when to inject) and how a representative operational profile (workload) should be designed [12].

An experimental comparison was made between fault injection and error injection. The authors note that no research work compares the failure symptoms obtained using source code faults and SWIFI injected errors. The investigation focused on two aspects: "the cost in terms of setup and execution time for using the techniques" and "the impact of the test case, fault type and error type on the failure symptoms of the target system" [13].

The authors conclude that fault injection is more accurate than error injection and test case had a large influence on the failure symptoms either for faults and errors injection.

Software Fault injection using representative fault models

This technique is based on the manipulation of source code, object code or machine code, i.e. original instructions are changed to other instructions. The aim of this method is to emulate common errors in software development.

Random fault injection is used in many software testing approaches [28, 39]. They replace the program data with random faulty data or introduce faults into random locations and then run test cases to validate if the software correctly handles the faults. Random fault injection, however, results in poor coverage and low bug-detection accuracy.

To solve this issue, some methods [22, 69] use program information in order to guide fault injection and generate efficient test cases.

Madeira et al. [37] presents a criteria for injecting code changes:

- **What to inject:** A set of program instructions are replaced by other instructions, based on common types of software faults found in real systems;
- **Where to inject:** As mentioned a fault is injected into the program code, it can be in the source code or in its binary executable. The code location is selected according to the type of fault. For instance, “a fault that affects variable assignments can only be injected in a code location containing a variable assignment” [44];
- **When to inject:** In order to reproduce the true nature of software faults, they should be injected into the target code before they are executed. Some papers presented by Natella et al. [44] inject code changes at runtime in order to force the occurrence of failures at a desired or random time.

The major adversity of this technique is fault representativeness. Ideally it is intended to emulate all the defects that a program may contain. However, if it were possible to know all its defects, they would be easily corrected.

Durães et al. [18] observed that a large percentage of faults falls on well-defined classes. Through a small set of emulation operators it was possible to emulate precise software faults.

3.3.4 Fault Injection Tools

Since fault injection is a mature concept, several fault injection tools have been developed over the years.

The information provided in Table 3.1 is based on previous reading of fourteen articles. Most of the articles do not explicitly mention if they are open-source and if they have a graphical interface, so besides reading them a deeper research was done.

Table 3.1 presents some characteristics of fault injection tools:

Tool name	Year	Fault model	Open-S.	GUI
MESSALINE [3]	1990	Stuck bits using HWIFI at the pin-level.	–	–
FINE [31]	1993	Source-code mutations emulating Software (SW) faults and stuck bits and bit flip (permanent and hardware faults).	–	–
FERRARI [30]	1995	Bit-flips using SWIFI.	–	–
FIAT [54]	1995	Bit-flips through SWIFI.	–	✓
FTAPE [64]	1995	Bit-flips and zero/set in CPU registers and memory and error codes in disk Input/Output (I/O). Uses SWIFI method.	–	–
Xception [10]	1998	Stuck bits and bit-flip emulate permanent and transient hardware faults. SWIFI approach.	–	✓
NFTAPE [60]	2000	Bit-flips, communication errors and I/O faults in distributed systems. Apply SWIFI method.	–	✓
GOOFI [2]	2001	Single or multiple transient bit-flip faults using techniques such SWIFI and Scan Chain Implemented Fault Injection (SCIFI).	–	✓
G-SWFIT [38]	2002	Code mutation at the machine code-level emulating software faults.	–	–
GOOFI-2 [58]	2010	Emulate transient hardware faults using single or multiple bit-flip errors . Test port-based technique used.	–	–
EDFI [22]	2013	Code mutations performing execution-driven fault injection.	–	–
FAIL [53]	2015	Single or multiple bit-flips in CPU registers and memory emulating transient or permanent faults. Support simulation-based fault injection and a hybrid technique between test-port-based Fault Injection (FI) and SWIFI.	✓	–
LLFI [35]	2015	Single bit-flips in CPU registers. A pre-runtime fault injection tool (SWIFI).	✓	✓
ProFIPy [14]	2020	Code mutations defined by the user emulating software faults.	–	–

Table 3.1: Fault Injection Tools.

- **Fault model:** refers to the fault model and technique that the tool implements;
- **Open-S. :** open-source, tool is or not available for public use;
- **GUI:** graphical user interface, tool with or without graphical interface.

There are several differences between the tools presented and others that do not appear in the table, such as fault models, the place where the fault is injected, the fault trigger, when to inject, what is intended to be evaluated (fault tolerance, fault recovery, etc.), among others.

3.4 Fault injection using failure models

Fault injection using fault models is often used to refer to failure models in various research works [28, 40, 41], however there are differences between both, namely in the fault models (Section 3.3.1). There are not many research work addressing the topic of fault injection using failure models which implies a smaller study compared to fault injection using fault models. The purpose of fault injection using fault models is to create a failure by making some sort of value corruption in the hardware or software of a system. On the other hand fault injection using failure models can be considered as forcing the system to fail, introducing the failure. Failure models can allow faster evaluations due to its quick and clear effects, whereas fault models can cause masked faults which do not lead to a wrong state, they do not cause failures.

Failure model is usually defined based on previous experience and human knowledge which may not take into account the real failures that occur in a system. Thus, these models are unlikely to be representative. For example, by injecting hardware faults using fault models the bit values of the CPU registers are changed, on the other hand, failure models aims to create characteristics of failures that may occur: crashes, disk failures, network partitioning [28]. Through software fault models modifies the source code of a program, mutations, whereas failure models can corrupt attribute values, method parameters or wrong return values [40]. Table 3.2 summarises the different types of models between fault and failure.

	Fault models	Failure models
Hardware	Stuck bits [3, 10, 31] Single bit flips [10, 30, 54] Multiple bit flips [2, 53, 58] I/O faults [60]	Crash [28] Disk failures [28] Network partitioning [28]
Software	Code mutations [14, 22, 31, 38]	Corrupt attribute values [40] Corrupt method parameters [40] Return wrong values [40]

Table 3.2: Differences between fault models and failure models.

failure models are most often used to evaluate distributed systems, cloud, or software systems, as they can easily emulate high level failures, which is often the main focus of works in these areas. In a system (computing nodes, communication interfaces, and networks), failure modes are used to model how faults affect different subsystems. A failure mode, then, describes the consequence of subsystem failures in a system. There are different failure modes including Byzantine failures, generalized as agreement problem [41], timing failures, omission failures, crash failures, fail stop failures and fail-signal failures [5, 6].

3.5 ucXception

ucXception¹ [11, 50] is a framework that allows to perform fault injection on a variety of target systems, from a local system to virtualized or cloud systems. ucXception allows the evaluation of hardware and software fault tolerance mechanisms and the dependability of the systems, combining a suite of fault injection tools capable of emulating hardware or software faults with different fault models (see Subsection 3.5.2).

It aims to reduce and simplify the effort in setting up a fault campaign for those who are less experienced in the field. The framework provides pre-made components and a base model that can be adapted according to requirements. It also allows a variety of metrics to be obtained to enable a wider and detailed analysis of the campaign.

Section 3.5.1 presents the main structure of the ucXception framework, mentioning the different components that constitute it. The subsection 3.5.2 discusses the fault tools present in the framework.

3.5.1 ucXception architecture

ucXception consists of a set of configurable elements according to the experience that the user wants to run. The use of this framework consists in designing a plan, set of campaigns. A campaign is a set of runs of a given campaign configuration. A run is the execution of the flow specified in the campaign. Runs can be executed with fault injection and/or without fault injection (golden runs).

Figure 3.3 presents a high-level overview of the ucXception architecture presenting core elements that can be changed by the user, such as:

- **Watchdog:** In order to ensure that runs do not exceed the user-defined allotment of time, a watchdog should be used to monitor and kill the workload application if the execution time exceeds the user's time limit;
- **Probes:** The probe is an application that is launched for the duration of the run to monitor and store information about the system or application being evaluated. There are two types of probes: pre-probes and post-probes,

¹ucXception GitHub <https://github.com/ucx-code/ucXception/>.

depending on whether they are launched before or after the workload has begun;

- **Pre-Probes:** Collect system-wide metrics;
- **Post-Probes:** Monitor specific processes.
- **Fault injection tool:** Injection tools implement a specific fault model (*e.g.*, a single bit-flip) emulating faults according to the model;
- **Validators:** The validators examine results obtained during runs and if the acceptance criteria have been met. If the validator fails then the run will not be saved to disk;
- **Parsers:** Parsers are used to convert the results of the run into a more valuable and compact format. These results are stored in the results Comma-Separated Values (CSV) file;
- **Transformers:** As the parsers an output is produced from raw input, however the output from transformers are stored as individual files in the run's own result's folder. Most commonly, transformers are used to convert raw data output from probes into a more manageable format, such as converting a binary data file from a resource monitoring probe into a CSV file.

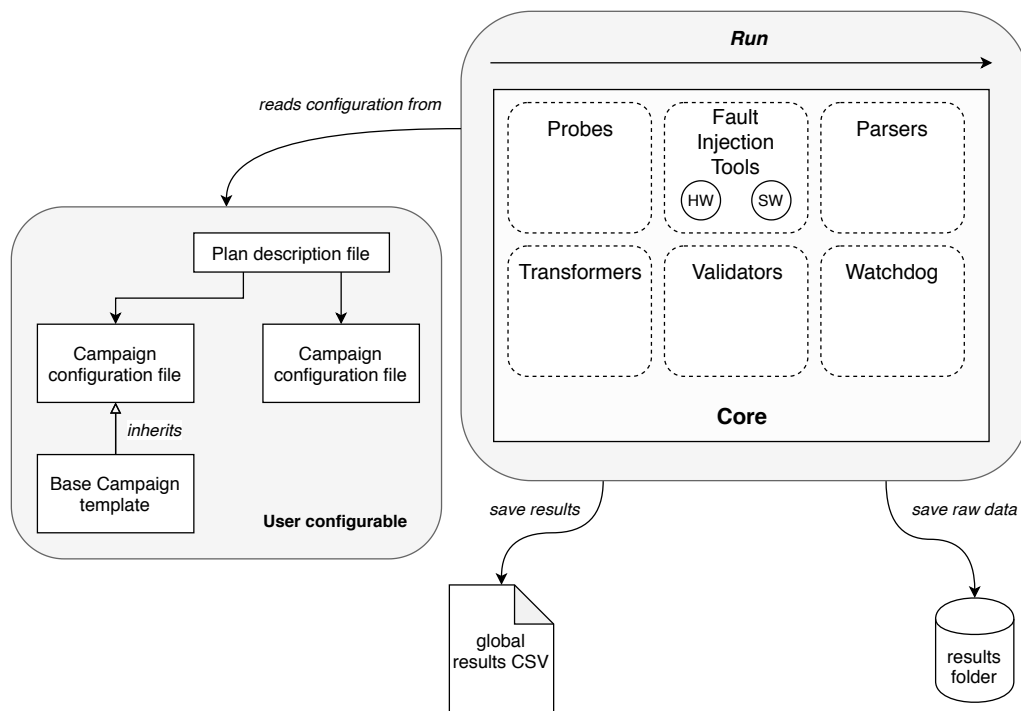


Figure 3.3: ucXception architecture [11].

The framework already includes some pre-defined components [11], as shown in Table 3.3.

As the framework allows faults to be injected either locally or in a distributed system, when setting up the campaign it is important to define whether we want

Component Type	Component name
Pre-Probes	<i>Logs probe / IntelPCM probe / Ping probe / SAR probe / TCPDump probe / Xentrace probe</i>
Hardware Injector	<i>ARM pinject / Intel pinject deadline / Intel pinject fp / Intel pinject v2</i>
Software Injector	<i>Injector</i>
Post-Probes	<i>Pidstat probe</i>
Validator	<i>Ensure Injection</i>
Parsers	<i>HW FI parser / SW FI parser / Pcap -> TCP parser / Info parser / MD5 output parser / Return code parser / Current folder parser</i>
Transformer	<i>Pcap -> TCP 2 CSV transformer / Pidstat 2 CSV transformer / SAR 2 CSV transformer / Ping 2 CSV transformer / Save output transformer</i>

Table 3.3: Pre-defined components.

the host to be remote or local. In order to define a remote host it's necessary to write the required information to perform login via Secure Shell (SSH) (Internet Protocol (IP), username).

Base template

A campaign configuration file defines each flow of the experiment run, but generally behaves as described by the base template. The default flow of an experiment run consists in the following ordered steps:

1. **Launch pre-probes:** Launch pre-probes. They start monitor their targets.
2. **Launch workload:** The workload start.
3. **Launch post-probes:** Launch post-probes. They normally require the Process Identifier (PID) of the process.
4. **Launch fault injection tool:** Launch fault injection tool. Performs only one injection per run to avoid influencing future injections. Despite the fault injection tool being launched at this point, the fault may be injected later, since the tool itself may have its own trigger mechanism. Normally the type of fault injected is chosen randomly and can be changed.
5. **Peak loop:** The workload executes and at some point during its execution, the fault injection will occur. Watchdog is launched to ensure that the workload finishes within the allotted time by the user.
6. **Post finish:** Consists in stopping the probes. It may include another operation if necessary.
7. **Extract data:** Stores the probe data in the run's results folder.

8. **Launch transformers:** Launch transformers that convert the stored data into a different format. placing it back into the results folder.
9. **Launch parsers:** Launch parsers and produce the output stored in the main results CSV.
10. **Launch validators:** Finally, launch validators to validate the correctness of the results.

3.5.2 Fault injectors of ucXception

ucXception contains three fault injection tools that implement different fault models. Two different fault injection tools to emulate hardware faults, one tool focused on a linux-based target system and the other on virtualized systems, and one tool for emulating software faults. In addition to the tools mentioned above it is possible to insert new fault injection tools.

Hardware faults in Linux-based systems

The tool is intended to emulate soft errors (see Section 3.3.1) affecting CPU registers or other CPU components (buses, Arithmetic Logic Unit (ALU), Float Point Unit (FPU), etc.) by implementing the single bit-flip fault model. The technique used is SWIFI as it is purely software-based, it does not rely on any hardware functionality and can be classified as a run-time approach as no modification to the target program's source is required.

A modern Linux kernel can run the tool as well as x86_64 and Acorn RISC Machine (ARM) architectures. The tool can be used in x86_64 systems as an injector for *rip*, *rsp*, *rbp*, *rax*, *rbx*, *rcx*, *rdx*, *cs*, *ss*, *ds*, *es*, *fs*, *gs*, eflags, and *r8* to *r15* registers and it also can be used to inject in FPU and Streaming SIMD Extensions (SSE) registers. It can inject into *sb*, *pc*, *lr*, *sp*, *ip*, *a1* to *a4* and *v1* to *v8* registers if executed on an ARM system.

The tool uses the *ptrace* functionality of Linux systems to be able to attach itself to the target process. It allows to access the values of the process registers and execute the bit-flip. It includes logging functionality that stores the exact timestamp of the time of injection and the register values before and after the fault injection.

The injection time can be defined in two ways: timeout and deadline. In timeout mode the user specifies how many milliseconds the tool should wait before executing the fault injection. In deadline mode the user specifies a UNIX timestamp, including milliseconds, that defines the desired moment of injection.

Initially the tool starts sleeping for a value defined by the user. When the timeout elapses the tool attaches itself to the process that intends to inject the fault. It extracts values from the process registers and prints the values to a standard output stream along with the current timestamp. Next it executes the bit-flip in the process registers and continues the execution. Finally it prints the new values from the registers. The flow can be seen in Figure 3.4.

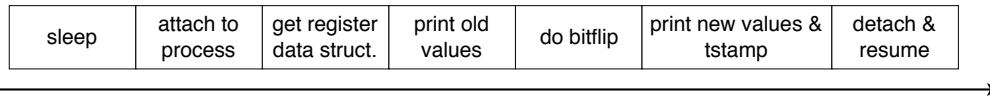


Figure 3.4: Hardware fault injection flow [11].

Hardware faults in virtualized systems

An additional tool for injecting faults is provided. It allows emulating hardware faults using the same fault model, single bit-flip in CPU registers and any of the *rip*, *rsp*, *rbp*, *rax*, *rbx*, *rcx*, *rdx* and *r8* to *r15* registers, however for use in virtualized systems. It can inject faults into any application operating inside a virtual machine. As long as nested virtualization is used, it can inject faults in hypervisor (i.e., virtual machine running inside virtual machine).

To implement the tool, modifications have been made to the Xen hypervisor that introduce a new hypercall and functions to control the fault injection process, along with modifications to the scheduling subsystem.

The injection process involves changing the value of the register in a data structure used to store Virtual Machine (VM)'s CPU state and which is updated before context switch. In this way, the injector makes use of the fact that the hypervisor must know the last CPU state between context switches of the VMs in order to inject faults. However, this means that the method is dependent on the frequency at which the context switches occur. If it is to perform a fault injection that affects only the hypervisor running on a VM, the tool provides a feature that can filter the application that will be targeted by looking at the value in the *rip* register.

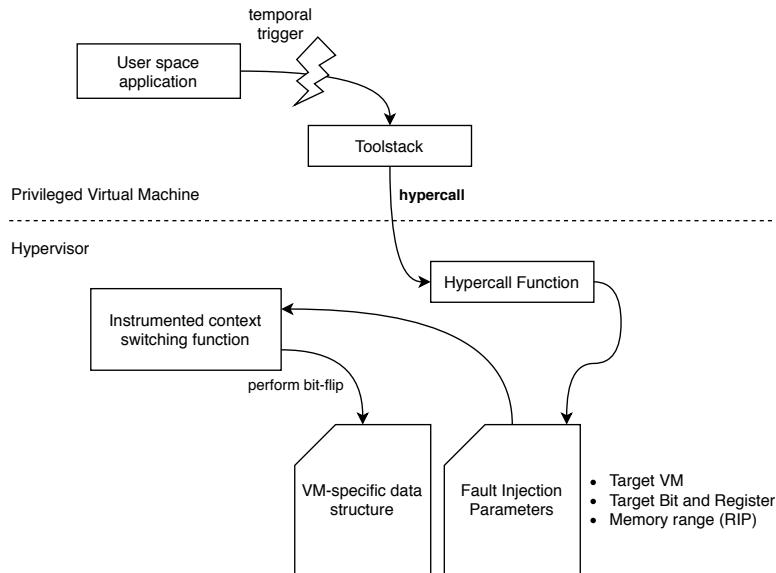


Figure 3.5: Fault injection flow for virtualized systems [11].

Figure 3.5 shows the flow in a graphical form and runs as follows: an application within the Privileged Virtual Machine (or the ucXception framework) provides triggering functionality, which is not embedded in the fault injection tool, and will call the toolstack at the right time. A hypercall is made from the toolstack

to a function inside the hypervisor, along with the parameters required for fault injection. The parameters are: the target VM (can focus in one specific VM if there is more than one), the target register and bit (place of injection) and an optional parameter defines the start and end of the memory range that the rip should be pointing at. All information is written by the hypercall function, which it will be read during context switching. As long as the VM that is receiving CPU time is the same as the target VM and its rip is inside the given range, the conditions are met and the bit-flip is performed before the VM starts its execution.

Software fault injection in C source-code

The third tool embedded in the framework applies code modifications (or mutations) to the source code in order to inject software faults. Faults are injected into programs written in the C programming language, following defined operators listed in Table 3.4. These modifications tend to represent software developers' mistakes [18].

Operators	Description
MFC	Missing function call
MIA	Missing if construct around statements
MIEB	Missing if construct plus statements plus else before statements
MIFS	Missing if construct an surrounded statements
MLAC	Missing and sub-expr. in logical expression used in branch condition
MLOC	Missing or sub-expr. in logical expression used in branch condition
MLPA	Missing localized part of the algorithm
MVAE	Missing variable assignment with an expression
MVAV	Missing variable assignment with a value
MVIV	Missing variable initialization with a value
WAEP	Wrong arithmetic expression in parameters of function call
WPFV	Wrong variable used in parameter of function call
WVAV	Wrong value assigned to a variable
WALR	Wrong algorithm – code was misplaced
WLEC	Wrong logical expression used as branch condition
EFC	Extraneous function call
EIFS	Extra if construct and surrounded statements

Table 3.4: Software faults operators [11].

The flow for software fault injection (see Figure 3.6) is divided in 2 phases: preparation phase and execution phase.

In preparation phase the software fault injector obtains the Abstract Syntax Tree (AST) from the input source-code through lexical and syntactic analysis. It traverses the AST to identify nodes where faults can be injected, according to the operators and their constraints [50]. Every node identified is modified in the tree, then converted to source code representation from which a patch file representing the fault is generated and stored in the filesystem.

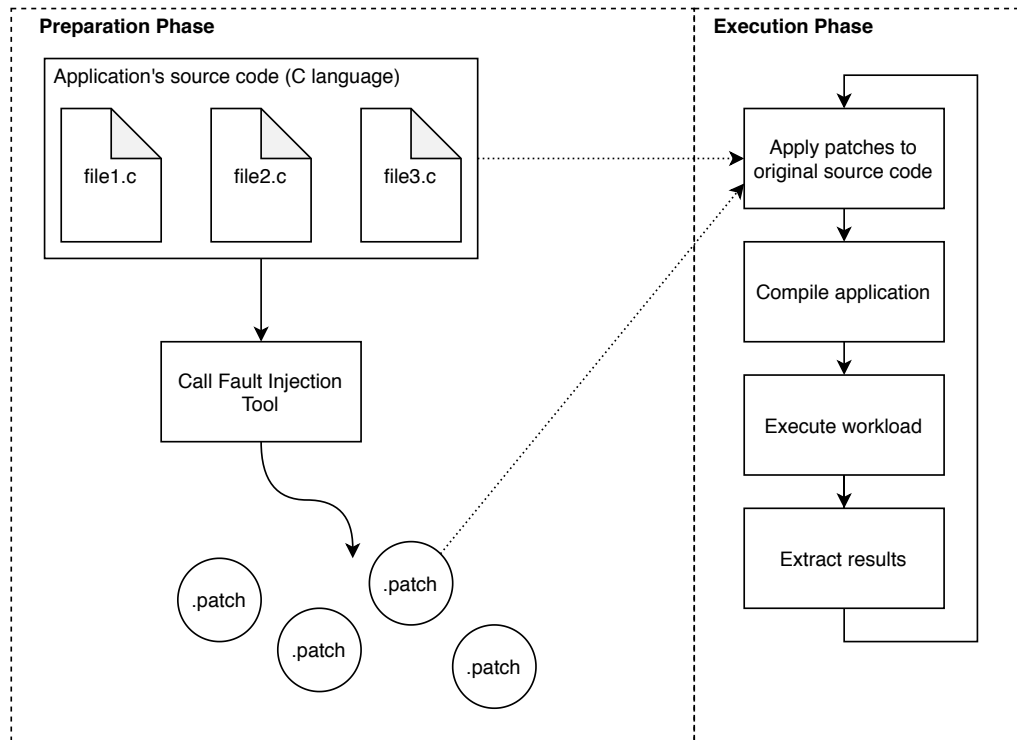


Figure 3.6: Software fault injection flow [11].

In execution phase each of the patch files generated will be applied once at a time and the compiled code with the patch will be executed according to the user-defined workload. At the end of running the workload for each patch applied metrics are stored.

3.6 Technologies

This section presents the study conducted on some of the most relevant technologies for both frontend and backend development. Section 3.6.1 presents the various technologies that were taken into consideration for frontend development. Section 3.6.2 discusses technologies for developing a backend module. Section 3.6.3 details the study conducted on the use of Docker containerisation technology.

3.6.1 Frontend technologies

In order to develop a frontend component, it is necessary to make an analysis of the various technologies currently available that allow the development of frontend applications in order to validate which technology best suits the purpose. From the various possibilities three tools were extracted, Angular.js, React.js and Vue.js for presenting a great relevance in the frontend development community, being used by well-known companies such as Google, Facebook, GitLab, among others, [29, 32, 63]. Table 3.5 presents a brief summary of the three technologies.

Angular

In 2016, Google created Angular to address the gap between increasing demands for technology and traditional strategies that produced performance. It is a framework based on TypeScript. Angular is the most mature of the studied frameworks, having a complete package with many features. Angular's learning curve is steep, and new developers may be put off by its concepts. It is a good choice for companies that have many developers and those that already use TypeScript [15, 26, 32, 63].

React

React was created at Facebook to address code maintainability problems caused by the app's continual inclusion of new features and adopts the JavaScript language. React can be considered as a mature technology and has a large number of contributions from the community. It is lightweight because it does not contain any inbuilt libraries, i.e. the programmers only import the necessary libraries and this allows some simplicity and flexibility for new programmers [15, 26, 32, 63].

Vue

Vue is the newest technology compared to the previous ones and like React adopts the JavaScript language. Vue runs on top of the Angular template and adopts React policies which makes it incredibly lightweight. Because it follows identical approaches to the two aforementioned technologies it has been widely adopted by the developer community. However, Vue's simplicity and flexibility has drawbacks, it allows poor code, making debugging and testing difficult [15, 67].

	Angular	React	Vue
Written in	TypeScript	JavaScript	JavaScript
Advantages	<ul style="list-style-type: none"> - Complete library - Highly testable - Strong community 	<ul style="list-style-type: none"> - Lightweight - Frequently updated - Good for beginners 	<ul style="list-style-type: none"> - Lightweight - Tiny and fast - Friendly to beginners
Disadvantages	<ul style="list-style-type: none"> - Steeper learning curve - Low performance 	<ul style="list-style-type: none"> - Requires library support - Complexities of learning JSX syntax 	<ul style="list-style-type: none"> - Limited community - Language barriers exist for plugins and components.
General community evaluation	Medium-High	High	High

Table 3.5: Comparison of frontend technologies.

3.6.2 Backend technologies

This section aims to show the various technologies available for developing a backend module. As the ucXception framework is written in Python it was important to find a technology that could integrate with it. The most popular technologies are Django REST framework, Flask-RESTful and Falcon [4, 8]. Table 3.6 presents a brief summary of the three technologies.

Django REST framework

Django REST framework is the oldest, most mature and complete toolkit to create a web Application Programming Interface (API). Developers can use the extensive and good documentation to learn how to work with the framework. Furthermore, this framework is trusted by popular organizations like Red Hat, Mozilla, and Heroku [4, 8, 16].

Flask-RESTful

Flask is newer when compared to Django. Developers use Flask Restful for creating an Representational State Transfer (REST) API quickly. With Flask, it takes only a few lines of code to get started making an API. It offers multiple data representations like Extensible Markup Language (XML), CSV, and HyperText Markup Language (HTML). The Flask API is easy to learn, and the documentation is excellent [4, 8, 21].

Falcon

Falcon provides a framework for creating high-performance, reliable, and scalable application backends and microservices. Using Hyper Text Transfer Protocol (HTTP) and REST architecture styles, Falcon creates a clean design. By providing a debugger for development, the REST framework simplifies the development process substantially [4, 8, 19].

3.6.3 Docker

Docker is an open platform for building, running, and managing containers on servers and the cloud. The Docker platform allows to package and run applications in a loosely isolated environment called a container. Security and isolation allow to run multiple containers simultaneously on the same machine. It is convenient to work on containers since they are lightweight, contain all the necessary components, and can be easily shared with others.

In order to understand and use Docker it is essential to know basic objects such as images and containers. The two objects can be defined as “an image is a read-only template with instructions for creating a Docker container” and “a container

	Django REST	Flask-RESTful	Falcon
Advantages	<ul style="list-style-type: none"> - Extensive and good documentation - Active community support 	<ul style="list-style-type: none"> - Very lightweight - Decorator for data formatting 	<ul style="list-style-type: none"> - Faster performance - It comes with debugger
Disadvantages	Lack of conventions and steep learning curve	May not perform too well under heavy load	Limited community
Community evaluation	High	High	Medium

Table 3.6: Comparison of backend technologies.

is a runnable instance of an image” [17], i.e., an image defines a container, as well as any configuration options provided when it is created or started. A *Dockerfile* is used to define the steps required to create an image and run it.

Docker Hub is a service provided by Docker to find and share container images with users. It works as an image repository and is one of the largest worldwide. Being very practical for both the publisher and the consumer, this service is used to publish the images created during the development of this task. Details of how it was used can be seen in the Section 5.4.2.

Docker architecture

A Docker architecture uses a client-server model (Figure 3.7) which consists of various components: Docker daemon, Docker clients, and the Docker Registry/Hub.

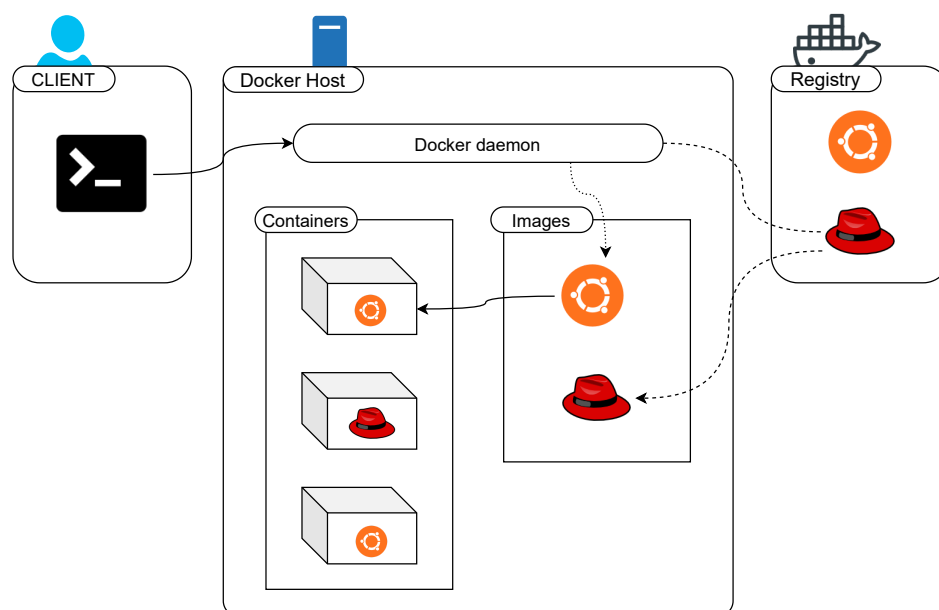


Figure 3.7: Docker architecture diagram [17].

The **Docker Daemon** manages Docker objects such as images, containers and volumes (persists data generated by and used by Docker containers), and does the heavy lifting of building, running, and deploying Docker containers. To manage Docker services, a Daemon can communicate with other daemons.

The **Docker Client** allows users to interact with Docker. If the Docker Daemon is running on a remote host, the Docker Client can connect to it remotely or reside on the same host. The Docker user can use commands on the Docker Client terminal to communicate with one or more Docker Daemon through REST API requests.

A **Docker Registry** stores Docker images. Docker registries are assumed to be repositories of images that can be stored and downloaded. There are two types of registries in the Docker: public Registry, include Docker Hub, and private Registry, normally used to share images within the enterprise.

Chapter 4

Requirements

One of the most important stages of the development process of a system is extracting requirements and the identification of user needs. Specifying correctly what the system should do and satisfy the needs of the users is fundamental to the success of the project. This process is the first step to be taken to improve the framework's usability.

Thus, the chapter presents the tools used for gathering requirements, starting with user stories (Section 4.1), then a complementary requirements gathering technique such as mockups (Section 4.2), define functional requirements (Section 4.3) and non-functional requirements (Section 4.4).

4.1 User stories

One of the aims of this work is to improve the framework's usability so that not only experienced users but also less experienced ones can use it without too much difficulty. It was decided to describe the functionalities of the framework according to the user's perspective. Through user stories it was possible to keep the focus on the users.

User Story (US) is a lightweight method for quickly capturing the "who", "what" and "why" of a product requirement. A software product must meet Acceptance Criteria (AC) in order to be accepted by a user. Thus, each user story is structured as follows:

US: As a <type of user> I want to <goal/objective> so that <benefit/result>

AC: Given that <type of user> when <a specific action is performed> then <expect some result>

User stories are organised into four modules: authentication, menu, campaign setup and campaign menu. Due to the large number of user stories, only a few user stories that are considered the core of each module are presented. The remaining user stories are exposed in the Appendix A.

1. Login into framework

- *US-2: As a user I want to login into my account so that I can access my campaigns and system functionalities.*
- *Acceptance Criteria: Given that I am a user when I write my credentials and click to submit then the system will verify my credentials and according to its response will allow me or not to login.*

2. List campaigns

- *US-5: As a user I want to be able to view my campaign history so that I can review past campaigns and their results.*
- *Acceptance Criteria: Given that I am a user when I click to view history then the system will display all my campaigns.*

3. Choose fault injection tool

- *US-10: As a user I want to choose which type of fault injection tool (e.g., SW, HW, Virtualized) to use so that I can focus in a specific fault type.*
- *Acceptance Criteria: Given that I am a user when I choose the fault injection tool then the system will change tool corresponding to the button pressed.*

4. Display data

- *US-26: As a user I want to analyse the results in an easily understandable format so that it is simpler and more straightforward to analyse.*
- *Acceptance Criteria: Given that I am a user when I select a finish or ongoing campaign then the system will provide a consolidated report of results.*

4.2 Mockups

A mockup is a representation of a product in action, a graphic element, and is often used for demonstration. It creates a clear description of what we want to create, helps visualize an end goal and allows to represent the functionality of a product in a visual way. The mockups were built based on the extracted user stories as shown in the Appendix B. In order to validate the mockups, two experienced users in the field of fault injection (one teacher and one PhD student that have used or are using fault injection and ucXception) gave their feedback. Through user feedback it was possible to improve aspects that were less clear or that were missing, thus the mockups presented are the final version. Table 4.1 presents the positive, negative and improvement aspects that were mentioned by the users.

A brief explanation is given about what can be observed in each mockup corresponding to the user stories presented above. Regarding the *US-2*, Figure 4.1

Positive	Statistics and chart pages are a very good approach for analysis; Download raw data; Use of breadcrumbs.
Negative	Unclear where to define workload parameters; Some misleading terms; It is only possible to upload files.
Improvements	Lack of some fields associated to components; Information popup explaining each component of the fault injection tool; Radio box is better than a dropdown for only two options; Being able to define a path to a file.

Table 4.1: User feedback.

shows the login page where users can enter the platform providing their username and password.

Figure 4.1: Login page mockup.

The *US-5* presented, which is depicted in Figure 4.2, shows a list of campaigns that belong to a logged in user. It displays some of the basic campaign information and allows to delete and cancel a campaign, as well as to filter the campaigns.

The *US-10* shown is associated with Figure 4.3, which exhibits the first campaign configuration page. The user can configure the campaign according to the requirements by setting up a campaign based on the chosen campaign type, upload the code/executable and configure the target host.

With reference to the *US-26*, Figure 4.4 shows basic statistics of a user selected campaign, such as the percentage of crashes, incorrect output, peak duration of golden runs and fault injection runs, among others. The chart allows the statistics to be tracked over the course of the runs.

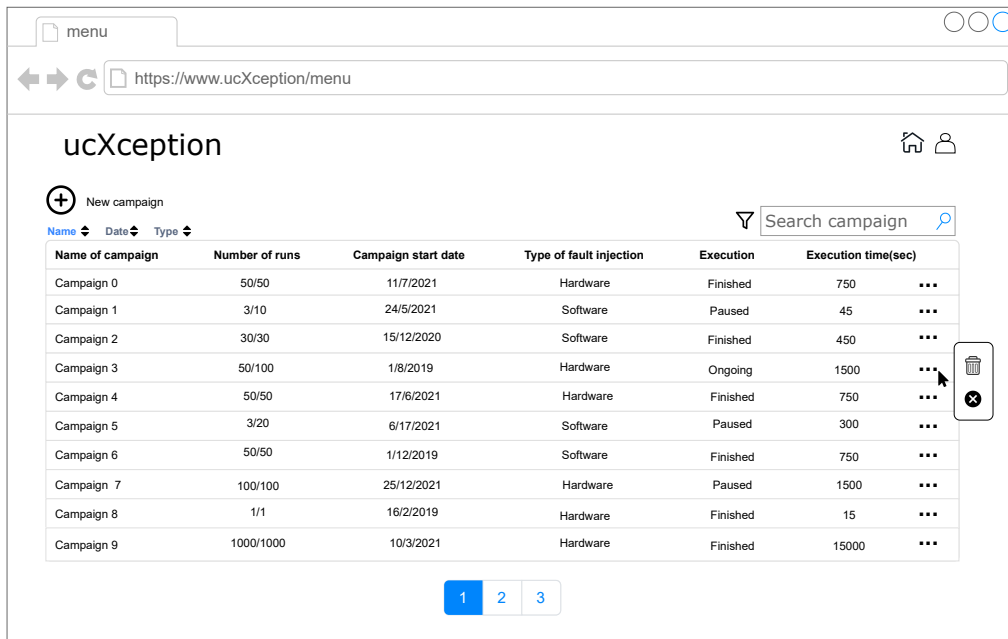


Figure 4.2: Menu page mockup.

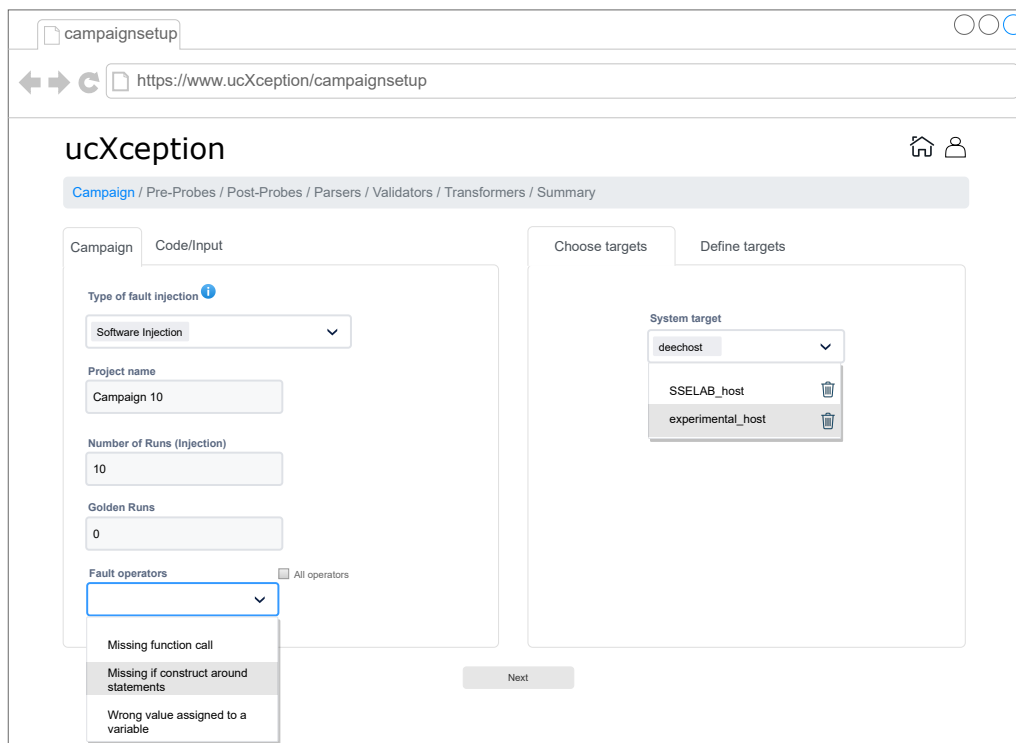


Figure 4.3: Campaign configuration page mockup.

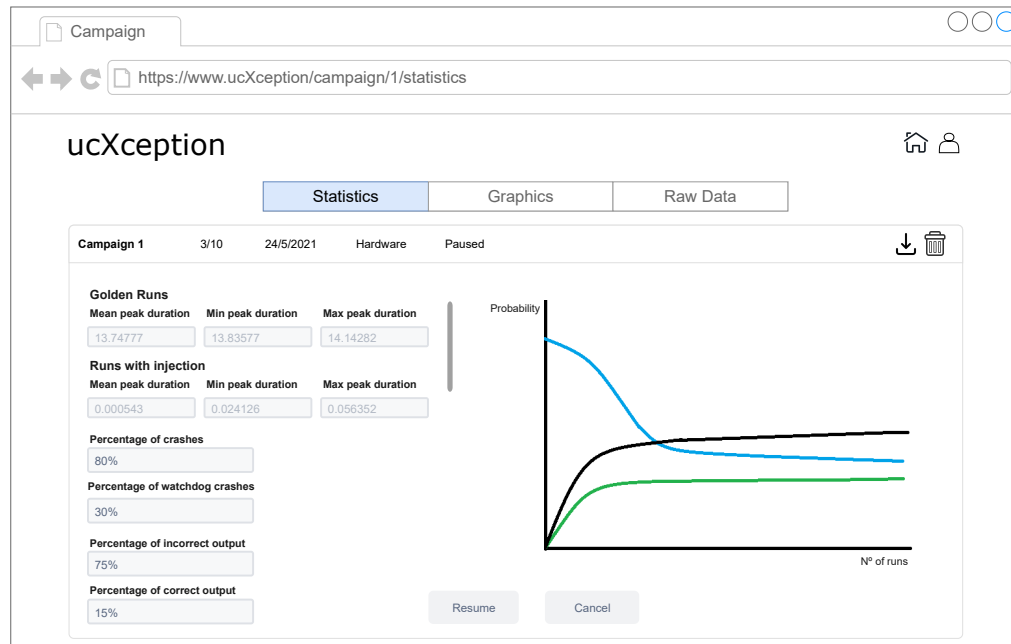


Figure 4.4: Campaign menu page mockup.

4.3 Functional requirements

Based on the user stories defined in Section 4.1, it was possible to outline all the requirements that the new framework should have in order to be considered a basic version of the product sufficient to appeal to consumers. For each requirement an evaluation was made according to its priority following the Moscow method, i.e. each requirement can present one of the following four types of priority:

- **Must Have (M)** - Defined as top priority, the requirements marked are necessary to complete the project successfully.
- **Should Have (S)** - Defined as medium priority, the requirements are essential for the completion of the project, but are not strictly necessary.
- **Could Have (C)** - Defined as low priority, when left out of a project, these requirements have less impact, but are still nice to have.
- **Won't Have (W)** - Defined as lowest priority, requirements are determined not to be a priority for the project timeline.

Table 4.2 presents the list of requirements for each module. Note that the implementation of the requirements implied modifications to the existing code of the original ucXception as well as the development of new code, for example for the graphical user interface.

Module	Requirement	Priority	User story
Authentication	Register	(M)	US-1
	Login	(M)	US-2
	Recover password	(M)	US-3
	Change password	(M)	US-3
	Logout	(M)	US-4
Menu	List campaigns	(M)	US-5
	Filter campaigns (pagination and filters)	(S)	US-6
	Order campaigns	(C)	US-7
	Delete campaign	(C)	US-8
	Cancel campaign	(C)	US-9
Campaign setup	Choose the fault injection tool	(M)	US-10
	Configuration of the fault injection tool	(M)	US-11
	Configuration of the watchdog	(C)	US-12
	Upload code and/or commands	(M)	US-13
	Create target/host	(M)	US-14
	Delete target/host	(S)	US-15
	Create execution	(M)	US-16
	Delete execution	(M)	US-17
	Create component	(M)	US-[18-21]
	Delete component	(S)	US-22
	Campaign summary	(C)	US-23
	Component summary	(C)	US-24
	Execute campaign	(M)	US-25
Campaign Menu	Display basic campaign data	(S)	US-26
	View statistics	(M)	US-27
	Build charts	(M)	US-28
	View campaign raw data	(C)	US-29
	Download campaign data	(S)	US-30
	Pause campaign	(C)	US-31
	Resume campaign	(C)	US-31

Table 4.2: Functional requirements of ucXception 2.0.

4.4 Non-functional requirements

Non-functional requirements, also called quality attributes, are characteristics that the system must have in addition to functionality. Quality attributes are difficult to identify and describe moreover have a great impact on the system architecture. They serve as constraints on the development of the system architecture. Quality attribute scenarios were used to describe quality attributes. The following example explains the description of a quality attribute scenario.

- **Source:** Entity that generates the stimulus (a human being, a computer system, or any other actuator);
- **Stimulus:** A stimulation situation requires a response when arriving at a system;
- **Artifacts:** Artifacts that can be a part or parts of a system which are stimulated;
- **Environment:** Certain conditions lead to stimulation. Depending on the situation, the system may be under overload or under normal operation;
- **Response:** Responding to a stimulus results in a particular activity;
- **Response measure:** In order to test the requirement, the response must be measured in some way when it occurs.

4.4.1 Security

Security is the ability of the system to protect data and resist unauthorised access [51]. The framework must maintain confidentiality, integrity and prevent outsiders from accessing other users' data. Users will only have access to their campaigns, i.e. the data associated to them.

Scenario 1 - Confidentiality

- **Source:** End user with access to the framework;
- **Stimulus:** Try to see other users' campaigns;
- **Artifacts:** REST API and Database;
- **Environment:** Normal/Fully functional;
- **Response:** Blocks access to campaigns that do not belong to the logged in user;
- **Response measure:** Probability of an attacker to discover the data.

4.4.2 Usability

Usability assesses how easy user interfaces are to use and also refers to methods for improving ease-of-use during the design process. Enabling efficient use of the platform and increasing the satisfaction and confidence that the user has when performing actions are two of the many focal points for maintaining good usability.

Scenario 2 - Increasing confidence and satisfaction

- **Source:** End user with access to platform;
- **Stimulus:** Create campaign;
- **Artifacts:** User interface;
- **Environment:** Run Time;
- **Response:** Positive or negative feedback. Change page content;
- **Response measure:** User interface response with latency of 0.1 seconds maximum [24], type of feedback received, user satisfaction.

A user always wants a response from the web application to confirm his actions. Through the study of Nielsen Norman [24] a response time in which the user feels that the system is instantaneous and that no feedback is needed beyond showing the result is 0.1 seconds.

Scenario 3 - Use system efficiently

- **Source:** End user;
- **Stimulus:** Wants to use system efficiently;
- **Artifacts:** User interface;
- **Environment:** Run time;
- **Response:** Distinct views with consistent operations (components like probes, parsers, etc, are configured in the same way);
- **Response measure:** Task time, number of errors, amount of time spent.

Consistency between pages allows the user to make fewer mistakes and apply the least amount of effort.

4.4.3 Extensibility

Extensibility is a measure of the ability to extend a system and the level of effort required to implement the extension [27]. Extensions can be performed through the addition of new functionality or through the modification of existing functionality. In this way, it is intended that it be possible to extend the number of existing fault injection tools in the framework by inserting them through containerisation technology with minimal effort.

Scenario 4 - Extending the system with new features

- **Source:** User;
- **Stimulus:** Add new fault injection tool to the framework;
- **Artifacts:** System;
- **Environment:** Build Time;
- **Response:** The new tool is added and can be used;
- **Response measure:** Effort of the addition, added without compromising the functioning of the framework.

4.4.4 Modifiability

The quality of being modifiable or capacity for modification is one of other aspects that is consider. The system must be able to tolerate changes, additions or removals of components, such as endpoints related to the REST API.

Scenario 5 - Change functionality

- **Source:** Developer;
- **Stimulus:** Modify endpoint;
- **Artifacts:** REST API;
- **Environment:** Build Time;
- **Response:** Changed inputs or logic from the functionality;
- **Response measure:** Effort of the amendments, change without affecting the functionality of the system.

Chapter 5

ucXception 2.0

This chapter presents ucXception 2.0, which is one of the main outcomes of this thesis. ucXception 2.0 represents a significant improvement over the original ucXception, namely, it adds a graphical user interface and enables a more practical, simpler and quicker configuration process.

This chapter begins by exposing the architecture of the framework based on the requirements collected (Section 5.1). Then the structure of the project is presented focusing on the organisation of the development of the framework (Section 5.2). This is followed by the modifications to the original framework (Section 5.3) and the considerations for encapsulating it (Section 5.4). Finally, the various functionalities developed are detailed (Section 5.5).

5.1 Architecture

This section describes the architecture of ucXception 2.0, comparing it to the original architecture, and listing the aspects that were taken into account during its planning. The purpose of this section is to show the decisions and reasons that led to the architecture planning and the changes between the original architecture and the ucXception 2.0 framework architecture. Moreover, the choice of technologies based on the needs shown in the architecture is presented and finally the entity relationship diagram is explained.

5.1.1 Original architecture

The original architecture of the framework is represented in Figure 5.1. It consists only of the Manager, which is the software that runs the campaigns to inject faults, communicates with the hosts and writes the results in a Comma-Separated Values (CSV) file. The user, in order to be able to use the Manager, must have previous knowledge about its structure and every new campaign must be created manually by changing the source code. In the architecture shown in Figure 5.1, it is presented how the Manager communication with a remote host is done. The

communication is done by Secure Shell (SSH) in which the Manager sends information related to the fault injection and receives information about the system state. All extracted data related to campaign executions can be accessed by the user, who then has to access the CSV file to perform its manual analysis.

Using the first version of ucXception was very difficult, complicated and required increased effort. There are multiple disadvantages associated to this framework:

- Pre-installation of libraries and technologies to be able to use the framework;
- Use of a CLI interface that is confusing and difficult to run campaigns for unaccustomed users;
- Need to know minimally well the structure of the python language and code. It involves a lot of effort to create new campaigns.

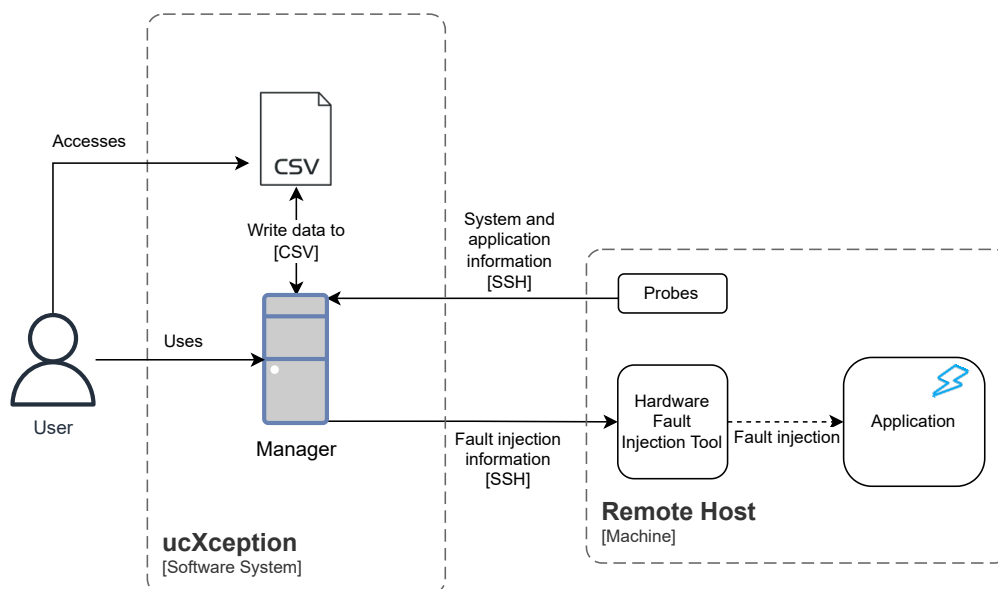


Figure 5.1: ucXception original architecture.

A new architecture was planned in order to make the framework simpler, more practical to use, reduce effort and based on the requirements raised previously. The new architecture is presented in Section 5.1.2 which clarifies how the various project elements will be organised and planned.

5.1.2 ucXception 2.0 architecture

ucXception 2.0 consists of two modules, the **Frontend** and the **Backend**, as shown in Figure 5.2. The Frontend provides the users a graphical web interface in which they will have access to the various functionalities of the framework, for instance, create a campaign or filter campaigns.

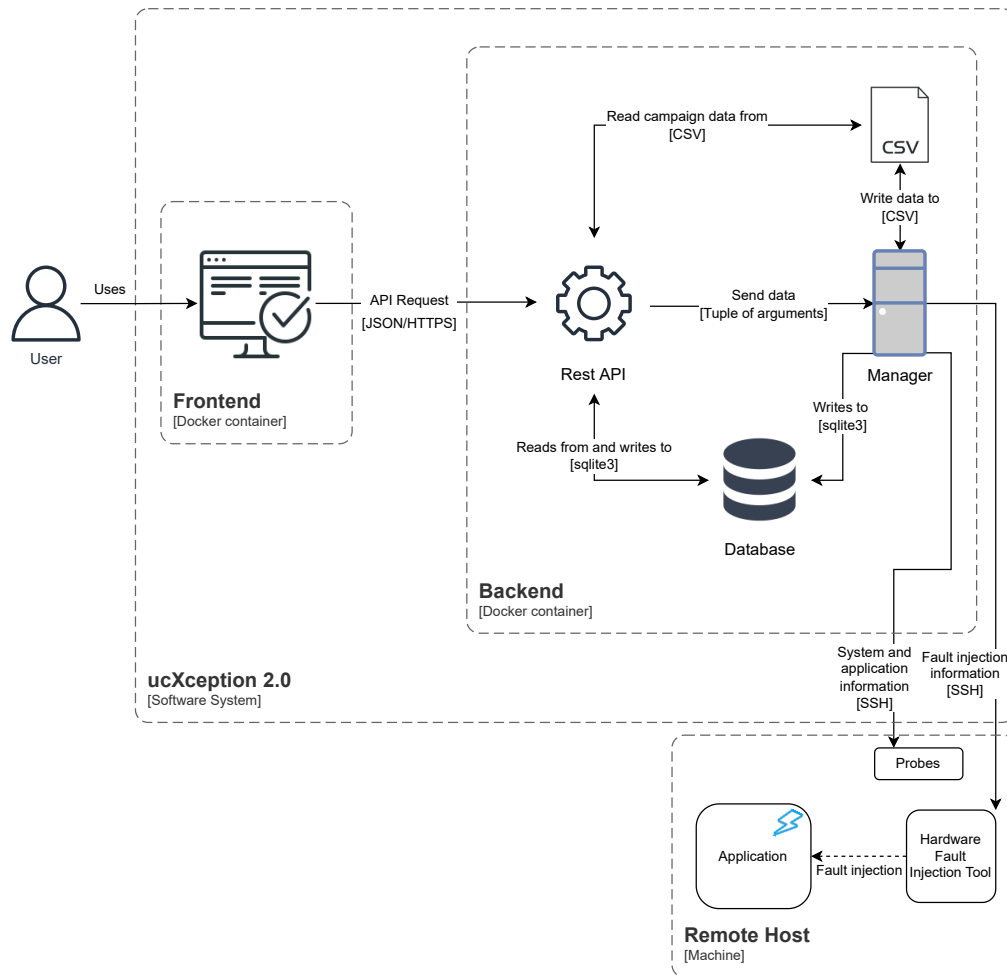


Figure 5.2: ucXception 2.0 architecture.

The Backend module consists of three software components: the Manager, a REST API and a database. The Manager is the heart of the framework and is the software component responsible for executing the fault injection campaign and storing its results. The Representational State Transfer (REST) Application Programming Interface (API) is an application programming interface that conforms to the constraints of the REST architecture style. The REST architecture simplifies communication between computer systems on the web by providing standards between them. The REST API exposes the functionalities of the Manager to the Frontend module. The decision to create a API was based on the fact that a controller for the Backend module was needed to manage and process the information and platform access, for example to handle campaign listing requests from a specific user or to handle access to the platform. The database stores user information, campaign configurations, components configurations, etc. Persisting this information is strictly necessary to perform various functionalities and to be able to maintain relationships between users and their data such that users have access only to their information.

The controller, API, has access to the database, in which all data received by the Frontend module will be stored and read. The Manager also has access to update the campaign status to indicate whether the campaign is still running or if it has

already ended. The Manager will update a row in the table that is only accessed when a new entry is created by API, so there will be no conflict between two or more entities trying to access the same table entry.

The Manager process is created by the controller and manages fault injection, i.e. it creates campaigns and injects faults into the system. While in the original framework we would have to run several instances of Manager manually for each experiment campaign, the ucXception 2.0 API spawns a process by creating a Process object and then calling its start method. This way whenever a campaign is required to run a new manager process will be created, allowing to have multiple campaigns running at the same time. No differences arise in Manager communication with a remote host between the initial architecture and the new one.

The controller reads the fault injection results from the CSV files generated by the Manager. The controller reads the corresponding file from a campaign and processes the data by creating new data in order for the Frontend module to display it graphically. Note that only the Manager writes the data collected after a fault injection in CSV files, this is the method used by the framework for the simple fact that it is faster and more organised. Only the name of CSV file associated with the user will be saved in the database.

The two modules, Frontend and Backend, are separated into two different containers, so that the user can configure them on different machines. Applying containerisation to the modules allows the modification and extensibility of the framework to be faster and simpler. Containerization is explained in more detail in Section 5.4.

5.1.3 Choice of technology for each module

Regarding the Frontend module, the features taken from the three technologies presented in Section 3.6.1 allowed for an easier and simpler analysis. Angular technology was put aside because it has a very steep learning curve despite the many features it contains and the trainee needs to learn it. The most difficult decision was between the React and Vue technologies as they were very identical. **The final decision was to use React**, since it was a language that had already been used several times by the trainee, knowing its limitations and characteristics better.

Of the three technologies presented for the Backend module, Section 3.6.2, Django despite being the most mature and most complete technology its learning curve is steep and as the project requirements may change it is not advisable to use this framework. Flask and Falcon are very similar in terms of performance and simplicity. **The decision was made to use the Flask** technology because it is more used and has more community support.

Regarding the choice of a technology for the development of the database, the client required the use of a technology that packs the entire database into a single file, SQLite [34, 52]. A decision of the client relies on fast and light engine, easy to

back up and low complexity.

5.1.4 Entity relationship diagram

The concept of Entity Relationship (ER) diagrams is to represent how "entities" such as people, objects, or concepts are interconnected within a system [36]. The ER diagram was used to design a relational database, as shown in Figure 5.3.



Figure 5.3: Conceptual diagram.

All the tables in the diagram have their own unique identifier, primary key. The table *user* is constituted by the user's personal data such as username, email, password and a public id. The table *reset_password* is used mainly for verification and validation of tokens generated when the user requests a password change. The table stores the expiry date and the token. The user can only request one token at a time, it is a one-to-one relationship.

The user can be associated to one or several campaigns and a campaign can be associated to only one user, one-to-many relationship. Each entry in the *campaign* table is composed of basic data such as a name, status, start as well as end date and the type of campaign. It also composes more essential parameters such as the path of the fault injection tool, the name of the CSV file and an identifier that indicates which hosts will be used for the target system and the fault injector target. Depending on the campaign it will be necessary to configure some of its parameters and define the system that will be evaluated. There are two essential tables for this purpose. The *campaign_parameters* table allows storing the user input as a blob for each campaign parameter. The name must be the same as the name of the corresponding campaign parameter. The *file* table allows storing two types of data, either a file or a path to the file as a blob. The *savedOnStorage* parameter indicates the type of value the user has chosen. The two tables have a one-to-many relationship, a campaign can have several parameters and files while these can only be associated with one campaign.

A campaign must contain one or more executions and hosts while these are only assigned to one campaign (one-to-many relationship). The table *executions* consists of the name of the execution, the parameter that indicates whether to inject faults, the number of runs to execute and the total number of runs executed at the moment. The table *hosts* contains a type, local or remote, and if the type is remote the domain and username parameters are used to define the remote host.

A campaign may have one or more components configured, but the component is only associated with one campaign, constituting a one-to-many relationship. The *components* table consists of a name given by the user and the name of the chosen component. It contains a relation to the *component_type* table which indicates the type of the component. Through a one-to-many relationship, an entry in the *component_type* table can be associated with more than one component. This is populated at API initialization by predefined values. A component to be configured it is necessary to set values for some of its parameters, so the *components_parameters* table stores these values as a blob. The name must be the same as the name of the corresponding component parameter. The component still has a connection to itself, one-to-one, because it may need another component to be able to work. Some components require a host to be defined so the *components* table contains a one-to-many relationship with the hosts table.

5.2 Project Structure

The organization and modular structure of complex and large projects is crucial to be as extensible and readable as possible. In this way it will be possible to modify code components that in a future enhancement or error correction of the system will not cause substantial delays, for example when fixing bugs. This section demonstrates how the organization was done in each module of the system, Frontend and Backend.

5.2.1 Frontend module

Regarding the structure of the web application, it was taken into account the separation of the various elements that form the various pages of the graphic interface. In order to organize it as modular as possible, the project was divided into three main folders: **views**, **components** and **utils**. Regarding the navigation between pages and route distinction the application incorporates two files: *route.js* which contains the defined paths for campaign creation, also essential for route distinction for the breadcrumb, and *routePages.js* which defines paths for authentication, menu and analytic pages. The *package.json* shows the dependencies/libraries installed and the *index.js* represents the starting point for the project to run.

The **views** folder contains the application views, i.e., code related to the construction of the page view presented to the user. The **components** folder contains some auxiliary components to the views, such as the header (top navbar), alerts, breadcrumbs and footer (bottom bar). Finally, regarding the **utils** folder, it includes

components such as modular creation of fill-in fields and files related to requests via REST.

5.2.2 Backend module

Regarding the Backend it is divided into five main folders:

- **api** - Folder associated with the development of the REST API;
- **fi-tools** - Folder related to fault injection tools;
- **framework** - All Manager development is present in this folder;
- **csv-files** - All CSV files generated by Manager are saved in this folder;
- **private-key** - Folder containing all the keys to establish a connection via SSH.

Most of the work was developing the API, however it was necessary to make some improvements and adaptations to the framework so that it could be managed by API. Firstly, the **api** includes a “main” file that allows the creation of the REST application. The various endpoints necessary for the development of the requirements were developed in blueprints allowing them to be modulated and separated from the application’s main file. It also contains the database creation file and respective Structured Query Language (SQL) commands which enable reading and writing actions on the database. Besides includes the *db* type file that corresponds to the database itself. The *settings* file defines configuration variables needed to define global aspects of the application, for example, the Uniform Resource Locator (URL) of the Frontend module, email server configurations, among others. When initializing the API, the configuration variables take the value of the environment variables that are set by the user in the *.env* file.

The **fi-tools** folder consists of several fault injection tools that already existed in the original version of ucXception such as hardware fault injection, software injection and virtualize hardware fault injection. The user can indicate the path to his own tool or he can use the fault injection tools present in the framework.

Regarding the **framework** folder, it covers all the components explained in Section 3.5.1, for example probes, parsers, among others. Whenever a user executes a campaign a process is created by API in which it performs the main function of the framework. This part of the framework suffered the most adaptations, as it needed to create campaign and component objects and process several elements associated to it using the information transmitted by the API.

Whenever a campaign has finished running, the CSV file generated by the Manager is saved in the **csv-files** folder. In this way the API accesses this folder to read the data when the user requests a data analysis. Finally the **private-key** folder allows the user to put a personal key into the project so that the container generated through containerisation can then use SSH communications with a remote host. This procedure is explained in more detail in Section 5.4.1.

5.3 Modifications to the original ucXception

One of the most important concerns regarding the framework is the way in which its various components and campaigns would be translated both for the graphical interface and for the API, i.e. to indicate that certain components and campaigns are built in a specific way. To this end, for each existing component and campaign in the framework, it is important that a JSON file with same name as the original is created. Since one goal of the framework is to allow users to use their own campaigns through it, the succeeding sections explain how these configuration files were and can be built by future users and in Section 5.4 explains how it can be added to the encapsulated framework. It is important to note that new components must follow this approach otherwise they can not be used.

5.3.1 Campaign configuration file

Initially the user must define an association name, the name of the original campaign file and the class name, as shown in Listing 5.1. These parameters are important so that the framework can differentiate between campaigns and components and also so that the Manager is able to instantiate them when building a campaign. Note that if these three fields are not present in the file, the user will not be able to use the campaign.

```
{  
  "campaign_name": "Sw fault experiment",  
  "campaign_file_name": "sw_faults_example.py",  
  "campaign_class_name": "SW_Faults_Example",  
}
```

Listing 5.1: Initial campaign configuration file example

In the campaign configuration file the user can also use the *configuration* and *parameters* fields, as shown in Listing 5.2. The *configuration* field allows to indicate campaign fields which require a path to a file, in which some required data must be indicated, such as those listed below.

- **name:** Name of the parameter to be displayed in the graphical interface;
- **regex:** Regex used to accept a given file;
- **require:** Indicates whether it is a mandatory field to fill in.

The *parameters* field is a dictionary that constitutes the parameters of the campaign. Each parameter within the dictionary is also a dictionary, which contains the information required for the validation of the parameter and follows the following structure:

- **type:** String, integer, multiple (accepts multiple values from the **values** list), single (accepts only one value from the **values** list);

- **default:** Default value that the parameter takes;
- **condition:** Depending on the type of parameter it accepts the following attributes:
 - Type is string:
 - * **minLength** - Minimum string size;
 - * **maxLength** - Maximum string size;
 - Type is integer:
 - * **minValue** - Minimum value;
 - * **maxValue** - Maximum value;
 - * **step** - Assist the graphical interface, if the user wishes to increase or decrease the parameter value;
- **values:** Field required if type is multiple or single. List where each position is a dictionary with a value and label (Listing 5.2);
- **require:** Indicates whether it is a mandatory field to be filled in.

```

{
  "configuration": {
    "app_path" : {
      "name": "App path",
      "regex" : "[a-zA-Z0-9_]+$",
      "require": true,
    }
  },
  "parameters": {
    "watchdog_dur": {
      "type": "integer",
      "default": 30000,
      "condition": {"minValue": 0, "maxValue": 100000,
        "step": 1},
      "require": true
    },
    "patch_files":{
      "type": "multiple",
      "require": true,
      "values": [{ "value": "MFC", "label": "MFC" }, {
        "value": "MIA", "label": "MIA" }]
    }
  }
}

```

Listing 5.2: Campaign configuration file example

Finally, a text excerpt can be defined to help explain the user more about the respective campaign, Listing 5.3.

```
{
  "information_help": "Campaign used for Software fault
    injection."
}
```

Listing 5.3: Information campaign configuration file example

5.3.2 Component configuration file

The initial part of the configuration file of a component is similar to the configuration file of a campaign, found in Section 5.3.1. Essentially what changes is the name of the key for each field. Note again that without these three fields the component cannot be used.

During the development of a component's configuration file the developer must pay attention to the parameters required for its creation. In total, four parameters are provided, *constructor*, *target*, *parameters* and *components_allowed*, in which the use of these parameters depends on the component to component as shown in the following list:

- **Probes:** *constructor*, *target*, *parameters*
- **Parsers:** *constructor*
- **Validators:** No parameters required
- **Transformers:** *target* e *components_allowed*

The case presented in Listing 5.4 depicts a probe component which indicates that it only requires three parameters as presented in the previous list.

```
{
  "component_name": "Probe Logs",
  "component_file_name": "probe_logs.py",
  "component_class_name": "Probe_Logs",
  "constructor" : ["local_dir", "wanted"],
  "target": {
    "display": "Target",
    "require": true
  },
  "parameters": {
    "local_dir": {
      "type": "string",
      "default": "",
      "condition": {"minLength": 0, "maxLength": 50},
      "require": true
    },
    "wanted": {
      "type": "multiple",
```

```

        "values": [{ "value": "xen", "label": "XEN" }, {
                    "value": "linux", "label": "Linux" }],
        "require": true
    }
},
"information_help": "A simple probe that extracts logs
                    from the target system during the Post finish
                    phase."
}

```

Listing 5.4: Probe component configuration file example

The *parameters* field is configured in the same way as explained in Section 5.3.1. The *constructor* indicates how the component is instantiated and varies between probes and parsers. Regarding probes, the constructor is used to indicate to the framework the order in which the parameters of the *parameters* dictionary should be ordered during the instantiation of the component. Regarding parsers, the *constructor* must be a string, as shown in Listing 5.5. The constructor string may contain the name of one of the parameters from the *parameters* dictionary, in which the framework replaces the parameter's name and the value assigned by the user when using the graphical interface.

```

{
    "constructor" : "[self.app_output, expected_len, '
                    expected_md5']"
}

```

Listing 5.5: Parser constructor example

The *target* indicates that the component can have an associated host. The user can provide a name for display in the graphical interface and can define whether or not the target is mandatory for building the component.

Finally, the *components_allowed* field is used only for transformers. This field allows to indicate which components the transformer accepts, Listing 5.6. *component_type* should be a component type specified in Section 3.5.1.

```

{
    "components_allowed" : {
        "require": true,
        "values": [
            {
                "component_type": "preprobes",
                "filename": "probe_sar.py"
            }
        ]
    },
}

```

Listing 5.6: Components allowed example

5.4 Containerization

One of the goals is to encapsulate the ucXception 2.0 framework which can then be made available so that anyone can easily and quickly configure it in their system, thus, this chapter aims to show the possible functionalities of the chosen technology and how it will be useful for this goal.

The technology used is **Docker**, as it is currently the most used, having an active community, good documentation as well as examples and is used throughout the development lifecycle for fast, easy and portable application development. Sharing the application and running applications on the same machine or on different machines are features that Docker enables with simple and quick configuration.

5.4.1 ucXception 2.0 containerization

Initially two *Dockerfiles* were created one to create the image for the Frontend module and another for the Backend module. First will be explained all the commands in order to build an image and later will be presented and explained in detail the respective *Dockerfiles* of each component.

Some of the commands used are as follows:

- **FROM** - Sets the base image for subsequent instructions;
- **COPY** - The command copies new files or directories from the current client directory to a destination path of the container file system;
- **RUN** - The instruction will execute any commands;
- **CMD** - Only need one instruction, if *Dockerfile* contains more than one then only the last instruction will take effect. Specifies which command to execute inside the container to be executed;
- **WORKDIR** - The command sets the working directory for any previously referenced statements that follow it in the *Dockerfile*;
- **ENV** - The instruction defines environment variables through a key-value pair or a file;
- **ENTRYPOINT** - Using this command user can configure a container that runs as an executable;;
- **EXPOSE** - During runtime, Docker receives instructions to expose the specified network ports.

Frontend containerization

At this point after a brief introduction to the basic commands to create a *Dockerfile* it is possible to make an analysis of the essential commands to create an image

of the Frontend module, represented by Listing 5.7. The comments present in the *Dockerfile* help clarify each command beyond the following explanation.

Initially the base image is defined, *Alpine*, this image was chosen because it is very light and the Frontend module does not need to make great computational efforts. Then the name of the working directory is defined and the essential modules for the execution of the app in *React*, the package files that contain all the necessary dependencies for the project and also the entire project are copied to this directory. The instruction in line 13 performs the installation of the dependencies previously copied. Subsequently it is defined which port will be exposed from the container. The last command executes the instructions given to run the app.

```

1  # pull the official base image
2  FROM node:alpine
3  # set working directory
4  WORKDIR /app
5  # add '/app/node_modules/.bin' to $PATH
6  ENV PATH /app/node_modules/.bin:$PATH
7  # install application dependencies
8  COPY package.json ./
9  COPY package-lock.json ./
10 RUN npm i
11 # add app project
12 COPY . ./
13 #expose port
14 EXPOSE 3000
15 # start app
16 CMD ["npm", "start"]

```

Listing 5.7: Dockerfile for the Frontend module

Backend containerization

In order to explain the Dockerfile of the Backend module, the Listing 5.8 is presented and as it was done in the Frontend module it will be explained in detail how this file was built.

The base image was first defined following the basic logic to create a *Dockerfile*. In this case *Ubuntu* was defined, because the framework was built based on this image. Then, some usefull packages for the framework are installed, such as python. The working directory is set and all the project content is copied. It is then installed some necessary dependencies for python, through a file.

One of the adversities encountered during the realization of the framework containerization was to find a way to allow the container to establish connection SSH with a remote host. Whenever a container connected for the first time to a remote host it implied the user to generate a key, i.e., it would not be practical for a user to enter the container's console and type his credentials to generate a key that would be allowed by the remote host.

In order to overcome this problem, the user should generate a key that is allowed on the remote host and then associate that key to the container's system-wide configuration file, as represented in line 13. The execution of that line will specify an identity file where its authenticity will be read. Line 14 gives the file permissions to be read. Subsequently it is defined which port will be exposed from the container. Finally, it executes the instructions given to run the application.

```

1   # pull the official base image
2   FROM ubuntu:18.04
3   # update and install essential packages
4   RUN apt-get update && \
5       apt-get install -y python3.8 python3-pip python3
6       .8-dev ssh rsync
7   # set working directory
8   WORKDIR /app
9   # copy project to working directory
10  COPY . /app
11  # install essential packages for python from a file
12  RUN pip3 install -r requirements.txt
13  # define
14  RUN echo "IdentityFile_/app/private_key/chave.pem" >>
15      /etc/ssh/ssh_config
16  RUN chmod 400 /app/private_key/chave.pem
17  #expose port
18  EXPOSE 5000
19  # start app
20  ENTRYPOINT ["python3"]
21  CMD ["call_api.py"]

```

Listing 5.8: Dockerfile for the Backend module

5.4.2 Environment setup

In order to publish the images to Docker Hub the following procedures were followed according to the Listing 5.9 sequence. First the image must be built using the *Dockerfile* file providing an image name and a version tag. Secondly define a label that will serve as the name for the repository and finally push the created image to the Docker Hub. Note that these 3 commands were used in both Frontend and Backend modules.

```

1   docker build -t [image:[tag]] .
2   docker tag [image:[tag]] username/repository:tagname
3   docker push username/repository:tagname

```

Listing 5.9: Environment setup pushing

Regarding the setup configuration, the user just needs to use the *pull* command to retrieve an image from the repository. The user has to do it from both images, Frontend and Backend. After that, the images are ready to use. In order to create

a container based on those images, the user simply needs to perform the `run` command.

Along with the `run` command it is mandatory to use the flag `-e` with a key-value pair, `REACT_APP_API_URL` or `FRONT_END_URL` and the url of the backend/frontend module. The flag `-publish` allows to set the address and port of the respective module. At the end of the command the base image is defined. The least important thing that is up to the user to decide is to set a name for the container, the `-name` flag allows to accomplish that.

```

1  docker pull pedroalmeida705/ucxception:webapplication
2  docker run -e REACT_APP_API_URL=http://[URL BACKEND
   ]:[PORT]
3  --name ucxception-api
4  --publish [IP ADDRESS]:[PORT]:5000 pedroalmeida705/
   ucxception:webapplication
5
6  docker pull pedroalmeida705/ucxception:framework
7  docker run -e FRONT_END_URL=http://[URL FRONTEND]:[
   PORT]
8  --name ucxception-web
9  --publish [IP ADDRESS]:[PORT]:3000 pedroalmeida705/
   ucxception:framework

```

Listing 5.10: Environment setup pulling

5.4.3 Extending ucXception 2.0 container

By studying Docker technology a way was found that allows to add new files to the framework, such as probes, fault injection tools (injectors), parsers, etc., in the containers or images through commands and not through the upload of those files by the Frontend module.

There are two possibilities, either the user only wants to insert in the container or also insert in the image. It is advisable to insert in the image since there may be some problem with the container and if it is necessary to delete it afterwards the user must do the procedures again to insert a file inside the docker.

First the user must insert the new file into the container. Through the first command shown in Listing 5.11 the user get as output detailed information, such as, container id, name, base iamge, etc, of all available containers. Then, second command, copies a file from a local folder to a given container, referenced through its id, to a given folder. Finally, if the user chooses to add the file to the image, simply use the third command to get the name of the image and its tag then execute the commit command which creates a new image through the changes in the container.

```

1  docker ps
2  docker cp [SOURCE PATH] [CONTAINER ID]:[DESTIANTION
   PATH]

```

```
3
4  docker images
5  docker commit [CONTAINER ID] [IMAGE]:[TAG]
```

Listing 5.11: Commands to add files to the container and image

5.5 Functionalities

This section will describe the functionalities developed for ucXception 2.0. The goal will be to present which functionalities each page of the web application contains and some implementation aspects. Table 5.1 shows an overview of the features that have been developed. The presentation will follow a logical order from the moment of authentication in the application to the analysis of a campaign's data. Regarding the number of endpoints, lines of code and files developed on the backend, there were 25 endpoints equivalent to 1500 lines of code and a total of 18 files. While in the frontend, 11 pages equivalent to 4500 lines of code and a total of 35 files were developed.

5.5.1 Authentication

The authentication module includes the functionalities related with registering a new account, signing in and recovering/changing the password. Features like having an administrative control of who is allowed to register or blocking registered users were not implemented as they were not considered as functional requirements of the project during the planning phase. Figure 5.4 shows 4 different page forms: login, create account, recover password and change password.

Starting with the create account form, the user needs to provide a username, an email and a password. The information provided by the user is checked syntactically by the API and, if it is correct, a field is created in the *user* table in the database, taking into account that the password will be stored in hash. A public id associated to the user is also generated. In any user action, the user is identified by its public id (which differs from the primary key of the table).

After creating an account, the user can enter the application providing the email and password. The respective endpoint to login checks if the data entered are correct and if there is a user with the credentials provided. If there is a user, a token will be generated with the public id, with a secret key and an expiry date of thirty minutes. Through the token the API is able to deny or allow the use of certain endpoints by checking if the user is registered.

As for password recovery, it was initially necessary to configure a service for secure testing of emails sent from the development environment, Mailtrap. Through the email provided by the user, the API generates a message with a link with a reset token and then sends it asynchronously. The generated reset token serves to verify the user and also has an expiry date like the API access token. It is stored in the *reset_password* table in order to be able to create a new verification

Module	Requirement	Priority	Completed
Authentication	Register	(M)	Yes
	Login	(M)	Yes
	Recover password	(M)	Yes
	Change password	(M)	Yes
	Logout	(M)	Yes
Menu	List campaigns	(M)	Yes
	Filter campaigns (pagination and filters)	(S)	Yes
	Order campaigns	(C)	No
	Delete campaign	(C)	No
Campaign setup	Cancel campaign	(C)	No
	Choose the fault injection tool	(M)	Yes
	Configuration of the fault injection tool	(M)	Yes
	Configuration of the watchdog	(C)	No
	Upload code and/or commands	(M)	Yes
	Create target/host	(M)	Yes
	Delete target/host	(S)	Yes
	Create execution	(M)	Yes
	Delete execution	(M)	Yes
	Create component	(M)	Yes
	Delete component	(S)	No
Campaign summary	(C)	No	
Component summary	(C)	No	
Execute campaign	(M)	Yes	
Campaign Menu	Display basic campaign data	(S)	Yes
	View statistics	(M)	Yes
	Build charts	(M)	Yes
	View campaign raw data	(C)	No
	Download campaign data	(S)	Yes
	Pause campaign	(C)	No
	Resume campaign	(C)	No

Table 5.1: Implemented ucXception 2.0 requirements.

and validation layer of the token to achieve greater security. If the user accesses the link, the user is directed to a page with the form shown in the lower right corner of the Figure 5.4. On that page he can enter a new password. When the user makes the request to change the password the token associated with the link and the passwords are checked. If everything is correct the password is hashed to the database, otherwise the password is not changed.

It is worth pointing out that each page of the web application, after a correct or incorrect action by the user, presents alerts that help or clarify the user. For each form it was necessary to create the respective endpoint in the Backend and the page in the Frontend. In total four pages were created, as shown in Figure 5.4, and six endpoints, listed bellow, were developed.

- POST: */register* - Create user account;

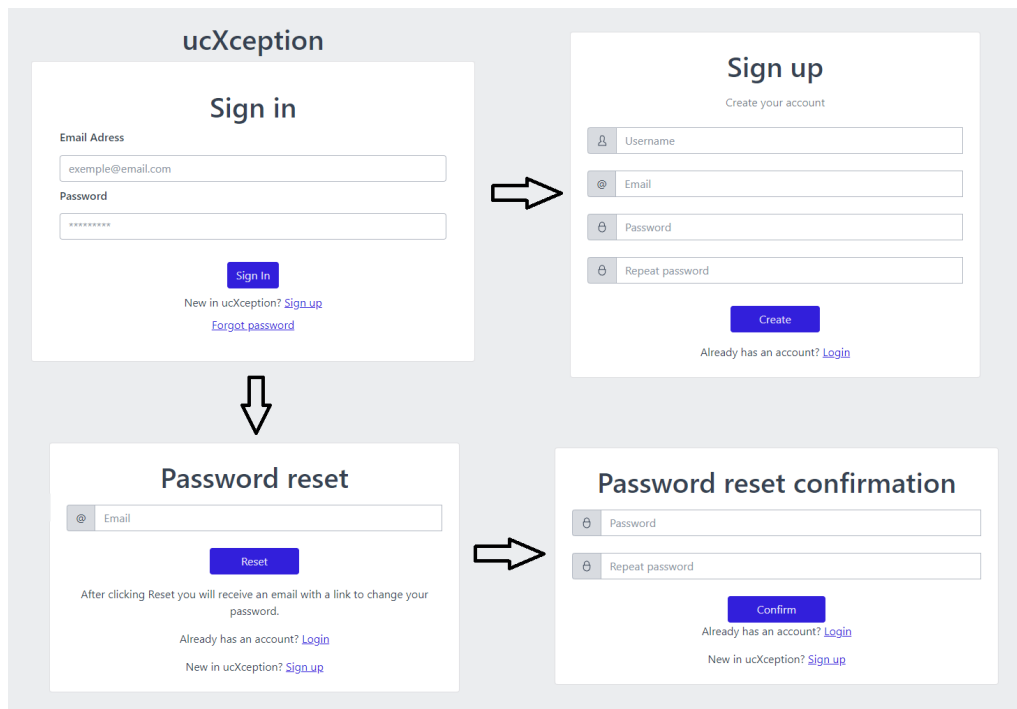


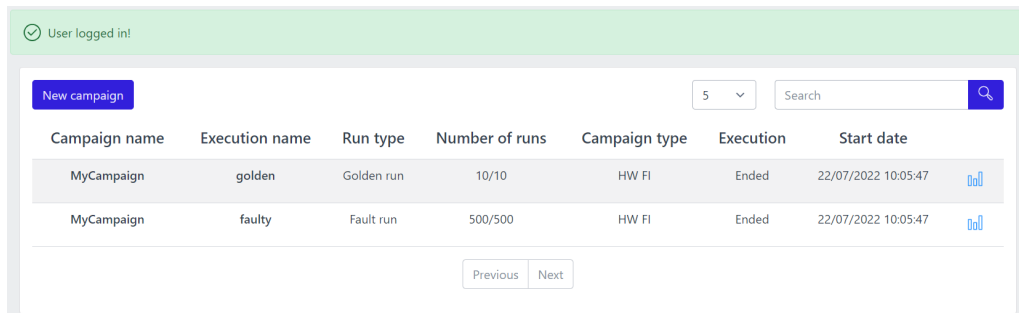
Figure 5.4: Login, register and password reset forms.

- POST: `/login` - Login to framework, generate access token;
- POST: `/verify_login_token` - Verify if access token is valid, basically to allow the pages to be displayed;
- POST: `/reset` - Create reset token and send email to user;
- POST: `/verify_reset_token/<reset_token>` - Verify if reset token is valid, essentially to allow the page to be displayed;
- PUT: `/reset/<reset_token>` - Change password.

5.5.2 Menu

The first page presented to the user after login is the menu page where the users can see basic information related to their campaigns, as illustrated in Figure 5.5. The user can filter campaigns by campaign name, execution name and campaign type name and change the number of campaigns to display at once. To accomplish this functionality only one endpoint is needed, `/campaigns` with method GET.

To view the campaign results in more detail, the user can click on the the button associated with each row of the table, i.e. each campaign execution. The button will redirect to the statistics page, Section 5.5.7. By using the “New campaign” button, the user is redirected to the campaign creation page, Section 5.5.3.



User logged in

New campaign 5 Search

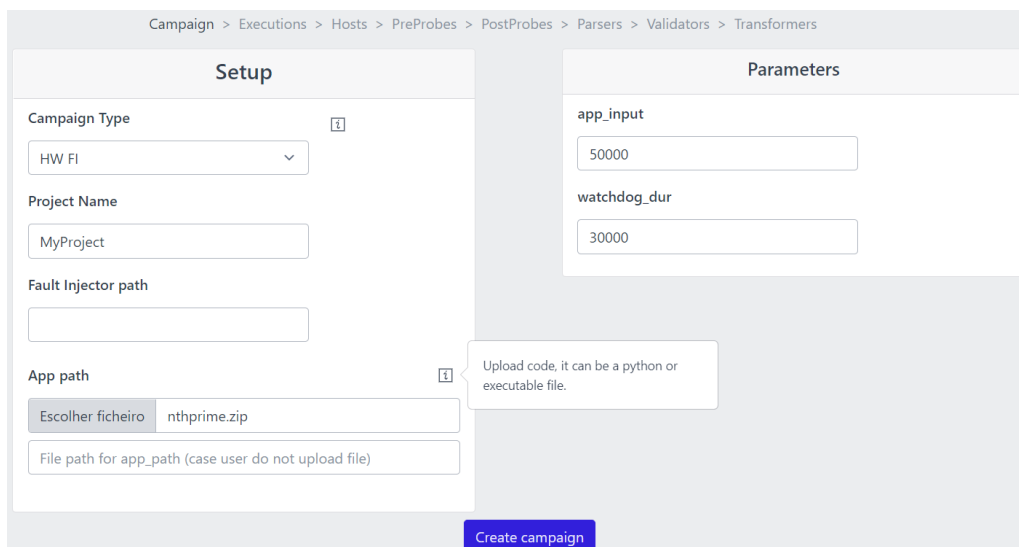
Campaign name	Execution name	Run type	Number of runs	Campaign type	Execution	Start date
MyCampaign	golden	Golden run	10/10	HW FI	Ended	22/07/2022 10:05:47
MyCampaign	faulty	Fault run	500/500	HW FI	Ended	22/07/2022 10:05:47

Previous Next

Figure 5.5: Menu page.

5.5.3 Create campaign

All data relating to the campaigns is received through an API endpoint, */campaigns/information* with method GET, in which it returns only the information associated with the campaigns. As demonstrated in Figure 5.6, two different types of fields are displayed in the campaign creation page, configuration and parameters. The configuration fields are associated to the general setup of the campaign, like name, path to the injector, files to upload, while the parameters are related to the more specific configuration of each campaign. Depending on the campaign selected, the parameters displayed change. To aid the inexperienced user to comprehend what is expected from each field, a help box which triggers a pop-up containing an informative message is used.



Campaign > Executions > Hosts > PreProbes > PostProbes > Parsers > Validators > Transformers

Setup

Campaign Type: HW FI

Project Name: MyProject

Fault Injector path:

App path: Escolher ficheiro nthprime.zip

File path for app_path (case user do not upload file):

Parameters

app_input: 50000

watchdog_dur: 30000

Upload code, it can be a python or executable file.

Create campaign

Figure 5.6: Create campaign page.

An endpoint was developed in order to create the campaign, */campaigns* with method POST. This endpoint is one of the most complex as it needs to perform several types of verification, syntax of the parameters, check if the type of the parameter corresponds to what is accepted by the campaign and perform verification of the files uploaded by the users. Through the information received by the setup fields a new entry is created in the *campaign* table. For each parameter that corresponds to the campaign, an entry is created in the *campaign_parameter*

table. The user can either upload a ZIP file or define a file path that will be stored in the *file* table.

5.5.4 Create execution

One of the most important configurations of a campaign are the executions. An execution defines how many runs will be executed and if faults will be injected or not. Thus, the user can create several executions with a varied number of runs. A table in Figure 5.7 is displayed to the user which shows all the runs created and allows the user to delete any undesired executions. The user cannot proceed with the creation of the campaign without any configuration of an execution, which will subsequently fail to run. In total three endpoints were implemented as can be seen in the following list:

- POST: `/campaigns/executions` - Create execution by adding a new entry to the *execution* table;
- `/campaigns/executions/<campaign_id>`
 - GET - Return all the executions from a specific campaign;
 - DELETE - Delete an execution from a specific campaign.

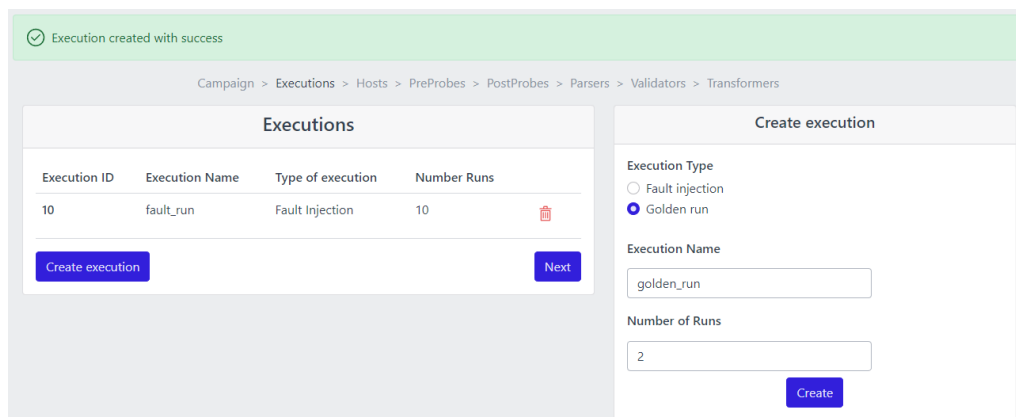


Figure 5.7: Create execution page.

5.5.5 Create host

A campaign needs to have a defined host either to run on or a target for the fault injection tool. Therefore, the user needs to choose whether the target will be used to run the campaign and/or will be to define the location of the fault injection tool. The user can choose a local or remote target, if choose a remote target then two fields are provided to fill in with the target details, like domain and username, if choose local the application automatically sets the value to local. The target configuration can be seen in the card to the right of Figure 5.8.

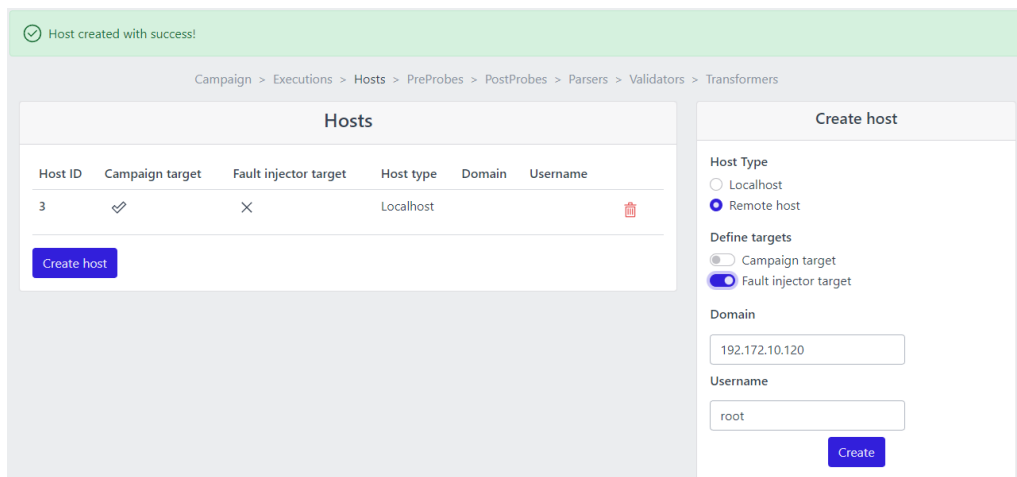


Figure 5.8: Create host page.

The user can configure multiple targets as some of the targets can be used for components, Section 5.5.6. The Figure 5.8 table shows a configured target and if the user has made a mistake configuring a target or does not want to use that target the user can delete it. Thus, to perform these features, there are three endpoints:

- POST: `/campaigns/hosts` - Create target by adding a new entry to the *host* table;
- `/campaigns/hosts/<campaign_id>`
 - GET - Return all the targets from a specific campaign;
 - DELETE - Delete a target from a specific campaign.

5.5.6 Create component

For each component a page is rendered with information corresponding to it, from pre-probes to transformers. Similarly to campaigns, all information related to components is returned by the API. The page is divided into 3 sections, as shown in the Figure 5.9. Each section and its essential endpoints will be explained, starting from user components to the configuration of a component.

The first section displays all the components that the user has already created. In Figure 5.9 no component had been created yet, while in Figure 5.10 there are already two components configured by the user. The endpoint, `/components/user/<campaign_id>` with method GET, is designed to return the *components* table by a particular campaign and a user.

The second section, which is the element positioned in the middle of the page, presents the various components that exist for each component type. The names of the various components are displayed using the `/components/<component_type>` endpoint with method GET, where the type of component must be specified. Through the radiobox the user can select a component to configure.

Figure 5.9: Create component page.

The third section displays fields for the user to configure the component, which is only displayed after the user selects the component. The endpoint `/components/<component_type> /<component_choice>` with method GET is used to obtain the information regarding these fields, in which it is necessary to specify the type of the component and the component chosen by the user.

However, there is an exception in the case of transformers, Figure 5.10, because there are components that depend on other components, i.e., if the user does not configure them previously, it will not be allowed to configure certain transformers.

Figure 5.10: Page to create a transformer that is not valid.

The endpoint `/components` with method POST was developed to create the components. The logic behind this endpoint is similar to the creation of campaigns, since it is also necessary to check that all the parameters received are in accordance with the chosen component. Initially a new entry is generated in the `components` table and for each parameter a new entry will be created in the `component_parameter` table.

5.5.7 View campaign statistics

After the campaign has been executed the user can access the results analysis page, Figure 5.11, from the menu. The statistics page presents an analysis of the results obtained by the execution of the campaign, such as information on the duration of the runs with and without fault injection and the various modes which have caused the failure, like incorrect output, percentage of crashes, among others. The chart allows the user to check the percentage of crashes in each run.

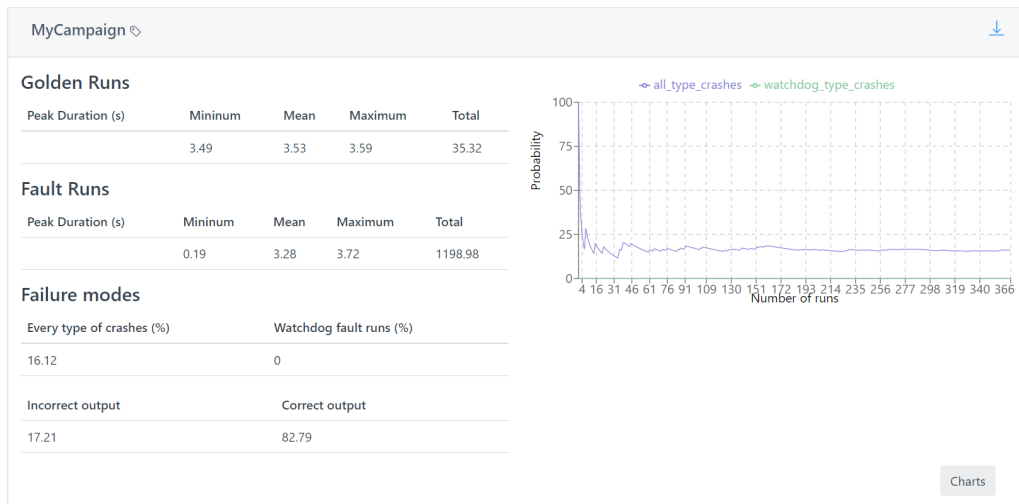


Figure 5.11: Campaign statistics page.

For this purpose an endpoint, `/campaign/<campaign_id>/statistics` with method GET, was implemented to return the information already processed. Through the `csv` file generated by the framework, the data processing is simple. It is basically calculated the minimum, maximum and average duration for the executions that fault injection occurs and does not occur, and the several failure modes are also calculated based on simple percentage counts. In order to create each line represented in the chart, an array of values is obtained from the same endpoint. There are two lines, one represents the percentage of the number of crashes of all types and the other represents crashes only by watchdogs for each run. Another type of analysis can be added.

The user can also download the results as a CSV file. This is implemented by an endpoint, `/campaign/<campaign_id>/download` with method GET, which with the campaign id it is possible to find out which file corresponds to the user. Note that it is verified if the campaign belongs to the user through the access token.

5.5.8 Build campaign charts

ucXception 2.0 provides the possibility for the user to create charts, Figure 5.12. It is currently possible to generate two types of charts, linechart and barchart, for any combination of values that have been extracted from the campaign `csv` file. It then leads to the first endpoint used, `campaign/<campaign_id>/columns` with method GET, which returns all the columns names in the file.

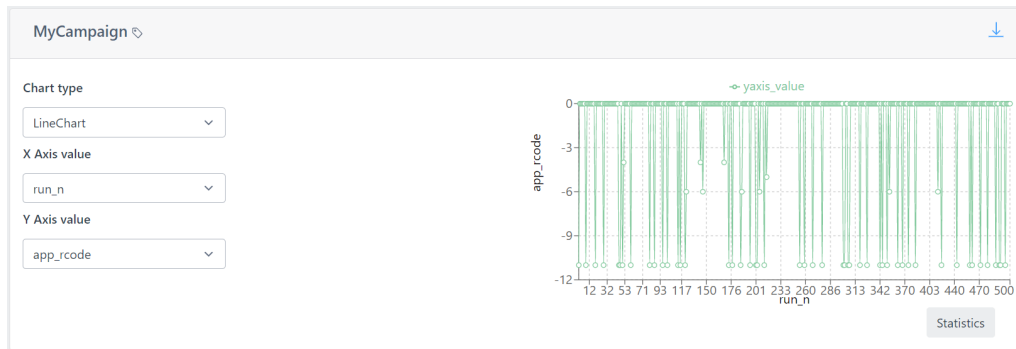


Figure 5.12: Campaign charts page.

The user can choose values for the x and y axes from the diverse number of possibilities. To change the chart, the user only has to choose a different value from the available dropdown boxes without having to click a button. The values of the chosen axes are returned via the endpoint `/campaign/<campaign_id>/chart` with method POST which for each chosen axis returns an array with the values of each row in the file.

Chapter 6

Testing

Software testing is one of the most important steps in software development, allowing to validate and verify if the product meets the expected requirements. During software development human errors can cause defects and failures so the testing process ensure that the software is defect free [49, 55].

This section focuses on the testing processes performed to the ucXception 2.0 framework. The first phase allowed testing the Application Programming Interface (API) since it is the core component and more susceptible to errors and, in the second phase, it was carried out usability tests, since the focus of the development of a new version of ucXception was to improve its usability.

6.1 Robustness testing

The first phase of testing conducted is to test the API regarding its robustness. Robustness means that the system deployed or under development is operating well in normal or ordinary conditions [9]. In order to test robustness, it was used a tool from another master's dissertation, EvoReFuzz - Evolutionary REST Fuzzer. It is a black-box tool for robustness testing of REST services using an evolutionary algorithm. EvoReFuzz takes advantage of the elements that make up a genetic algorithm, such as parent selection, crossover, mutation, fitness function, and elitism, to generate valid and invalid requests that will be sent to the system under test in an intelligent way.

Before setting up to run the tests, the tool required creating an OpenAPI specification. RESTful APIs are exposed with OpenAPI via a standard interface, allowing humans and computers alike to discover and understand their capabilities without having access to source code or documentation [61]. The OpenAPI specification was built using a library for Python and Flask that allows to simplify this process, called APIFlask¹.

The configuration of the framework and the EvoReFuzz tool were in different domains, allowing remote access to the application to be tested through a practi-

¹APIFlask Documentation in <https://apiflask.com/>.

cal case. The setup was relatively straightforward due to the application of containerization to the framework allowing this process to be practical and effortless. It was only necessary to start the Backend module container and configure external router ports, not docker related, to be accessible by other domains. Table 6.1 presents the environment of the experiments conducted.

Component	Description
Operating system	Windows 10 Pro 10.0.19043
CPU	AMD Ryzen 5 3600 6-Core Processor @ 3.59 GHz
RAM	16 GB
Disk	500GB BLUERAY SSD
Disk read speed	3000 (MB/s)
Disk write speed	1400 (MB/s)

Table 6.1: Experimental setup specification for robustness testing.

The requests generated by the tool are based on the OpenAPI specification. So, for example, in case the accepted value is a string the injection of a faulty request will be a value with random characters, special characters or even Structured Query Language (SQL) injection expressions in order to create a failure in the system. The test executed about 1200 requests per operation, i. e., for the 23 endpoints tested, 27600 requests were generated, which took about one hour and thirty minutes. Table 6.2 shows some of the requests generated by the EvoReFuzz tool that caused the system to fail.

	Endpoint	Body
C1	POST /campaigns/executions	campaign_id: -19 n_target_runs: -40894399 name: "2LLuqBq0KI" type: false
C2	GET /campaigns/executions/-21	
C3	DELETE /campaigns/executions/5	
C4	POST /campaigns/hosts	campaign_id: 32 campaign_target: true fault_injector_target: true type: "GWyw"
C5	GET /campaigns/hosts/1042	
C6	DELETE /campaigns/hosts/-63	
C7	GET /campaigns?page_size=-532& search-bar=3eoxSl &page=1155275	
C8	POST /reset	email: oJV}:0]6[,"Rr}5/"}
C9	GET /campaign/3/download	
C10	GET /components/HaAl9hRxzqzpAvR	
C11	GET /components/YRjZL6Ca/qH3Z8pBz	
C12	GET /components/yks0f55/aeL5C/3	

Table 6.2: Example of cases where the API has failed.

The analysis was done manually and essentially looked for operations that ended with a status code of 500. This event was noticed in cases C9 to C12 whenever the API tried to access a position of a dictionary that did not exist and lacked any protection. The operations that gave status code 200/202 were also analyzed with the purpose of verifying if there was a missing validation or verification, therefore, it was possible to detect errors in cases C1, C3, C4 and C6. The remaining cases were detected when examining other types of status codes, i.e. it was checked if the response matched the base response of API. The problems detected were also due to the lack of verification and validation. As shown in Table 6.3 the most common type of errors found were the missing validation of path parameters and body parameters as well as the missing verification of the existing elements in the database. It was also possible to analyze some status codes that were not adequate to the type of error that originated. Note that all the detected errors ended up being corrected to not compromise the operation of the API.

Case	Problem
C1/C4	Bad validation of positive numbers; Missing verification if the campaign with the given id existed; Missing verification if the campaign with the given id belonged to the user.
C2/C5	Missing validation of the campaign id.
C3/C6	Bad validation of positive numbers; It always returned status code 200 regardless of whether the element existed in the database; Use of body in delete method is not good practice.
C7	Wrong validation of positive numbers.
C8	Missing validation of the user email;
C9	Missing verification if the campaign with the given id existed;
C[10-12]	Missing validation and verification of the path parameters;

Table 6.3: List of problems encountered for each case.

6.2 Usability tests

As improving the usability of the framework was one of the goals, it was essential to plan tests to validate it and get feedback from potential users. The goals of usability testing are to identify problems, discover opportunities for improvement and learn about target user behaviour and preferences [42]. A usability test plan was prepared after robustness testing had ensured that the API was as bug-free as possible. This test will also allow discovering possible bugs in the web application.

A total of 5 participants was chosen, as sources such [45, 56] indicate that only 5 participants are needed to discover most problems and from that number the researcher would start seeing the same problems over and over again, wasting time. The chosen participants who were considered as potential users of the

framework were software engineering as well as electrical and computer engineering students, thus allowing the focus to be on the target user.

6.2.1 Test procedure

The tests were carried out remotely to avoid taking up too much of the participants' time and to be more convenient for them. The remote tests provided the opportunity to test Docker's configuration for remote accesses, ending up with no problems regarding its configuration and functioning. The tests took on average about half an hour. Each test started with a description of the ucXception framework, its disadvantages and an explanation of the new version. Subsequently, some concepts that participants may not be as familiar with were explained and participants were further asked for their consent to have the process recorded for later analysis. In the analysis phase the time taken by participants to perform the tasks and the number of clicks made were measured. In order to analyse the time of each task in more depth, a larger number of participants would be necessary, but it was decided to do this analysis as a complement.

Each participant was asked to perform eighteen different tasks:

1. Register an account;
2. Login;
3. Create a campaign;
4. Create executions/runs;
5. Create remote and local targets/hosts;
6. Remove the remote host;
7. Explain the *ucXception SW Parser* component;
8. Configure components *ucXception fi parser* and *App Returncode*;
9. Configure the *SAR 2 CSV* transformer;
10. Execute the campaign;
11. Search for a specific campaign;
12. Indicate the status of a specific campaign;
13. Indicate the average and total duration on faulty runs of a campaign;
14. Indicate the probability of crashes in run number 7;
15. Indicate the duration of the run 15;
16. Download the results;
17. Go to menu;

18. Logout.

After all the tasks have been completed, the participants were asked to fill in a questionnaire about the experience and usability of the framework. The participants after the test asked to continue using the application not only to see if they could find any bug but also because they were interested.

6.2.2 Test results

Table 6.4 shows the time taken from each of the participants and Table 6.5 shows the performance from each of the participants in the usability tests. For each task an expected value for its accomplishment is presented, so comparing the values obtained from each participant, most of the tasks performed presented satisfactory results. The number of clicks expected are the required to perform each task and the time presented is the best possible scenario, i.e. the time was calculated by the researcher. If the times deviate too much from the expected values, it can be assumed that the usability of some component has problems because participants were not able to perform in a timely manner.

The cells in the yellow coloured tables, indicate that there was a slight deviation from what was expected, however this is not directly related to the usability of the components associated with the task, but rather due to errors made by the participants or the fact that they took longer to perform the tasks because they did not understand the task well.

Task	Expected	User 1	User 2	User 3	User 4	User 5
T1	23	24	21	25	22	13
T2	12	14	14	15	14	15
T3	30	32	29	39	37	38
T4	28	30	32	26	35	22
T5	28	36	32	34	32	23
T6	3	3	3	3	3	3
T7	30	60	196	181	37	29
T8	20	23	17	28	23	23
T9	10	26	12	28	6	7
T10	2	2	2	2	2	2
T11	8	8	8	16	7	7
T12	8	10	9	19	8	9
T13	20	16	23	27	40	25
T14	6	7	7	13	12	6
T15	30	110	75	93	60	60
T16	2	2	2	2	2	2
T17	3	5	3	5	5	3
T18	3	3	3	3	3	3

Table 6.4: Time taken for each task performed by the participants.

Task	Expected	User 1	User 2	User 3	User 4	User 5
T1	6	6	6	6	6	6
T2	3	3	3	3	3	3
T3	8	9	8	9	8	9
T4	9	9	9	9	13	9
T5	10	10	10	10	10	10
T6	1	1	1	1	1	1
T7	5	13	21	19	5	5
T8	6	6	6	6	6	6
T9	3	3	3	7	3	3
T10	1	1	1	1	1	1
T11	2	2	2	2	2	2
T12	1	1	1	1	1	1
T13	3	3	3	3	3	3
T14	3	3	3	3	3	3
T15	5	12	7	12	6	7
T16	1	1	1	1	1	1
T17	1	2	1	2	2	1
T18	2	2	2	2	2	2

Table 6.5: Number of clicks on each task performed by the participants.

Other cells are coloured red, indicating a discrepancy with the expected values, which may be related to usability problems. With regard to the cells in red there are two tasks that stood out negatively, T7 and T15. Participants showed difficulties completing task T7 due to not being able to understand which page they were on, allowing to understand that the problem was because of poor breadcrumb clearness. Regarding task T15, participants have difficulty understanding which page and what inputs to provide in order to complete it. In task T9 some participants as they had not yet understood the functionality of breadcrumbs had difficulty completing this task as it had some similarities with task T7. Lastly, in task T17 there were no difficulties, but the participants tended to click on the top left of the page, corresponding to the logo of the framework, rather than going to the symbol of a house to return to the menu.

Post-Questionnaire

Following are some of the most relevant questions from the online questionnaire that participants were asked to complete in the usability tests. One of the questions asked to the participants was whether they had ever used any type of framework that injected faults and the majority of responses were that never used one, so this question was ignored because it was not possible to make a comparison regarding the usability of the other frameworks.

An analysis of the graphs shows that the overall experience of the framework was good, however in the Figure 6.3 there were two scores 3 where participants were keen to explain. The participants criticised that the most recent campaigns should

Do you think it is important for this type of framework to have a graphic interface without having to work directly with a command line?

5 respostas

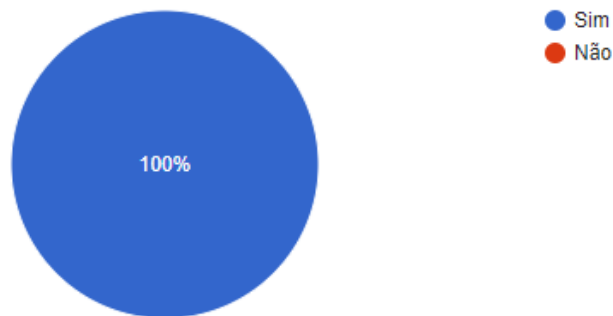


Figure 6.1: First question.

How would you describe your overall experience with the product?

5 respostas

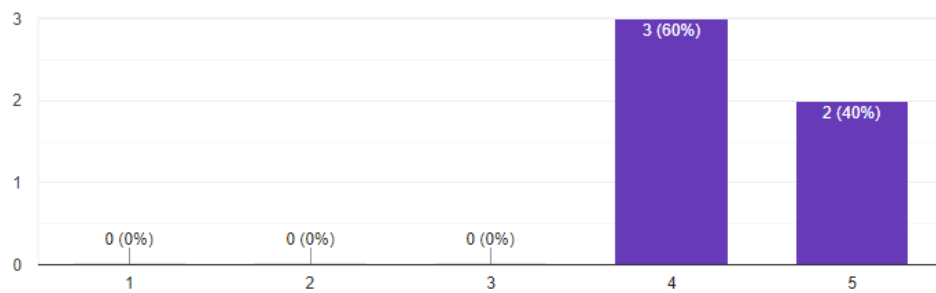


Figure 6.2: Second question.

How would you describe the consistency in the organisation of the various elements presented on the different pages?

5 respostas

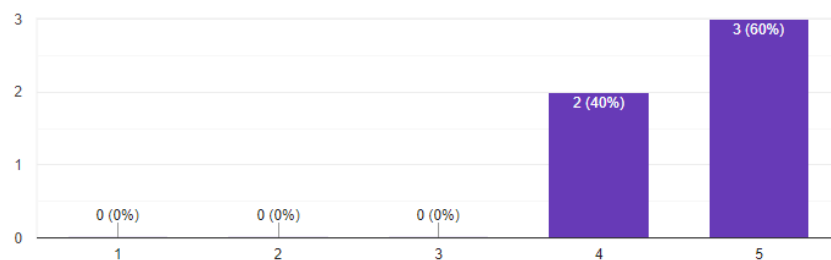


Figure 6.3: Third question.

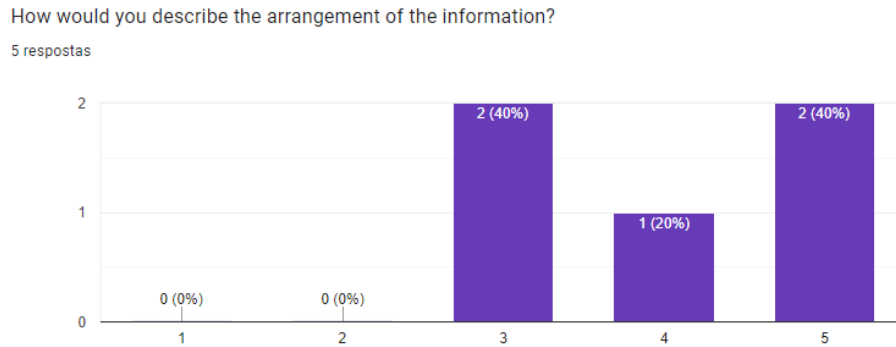


Figure 6.4: Fourth question.

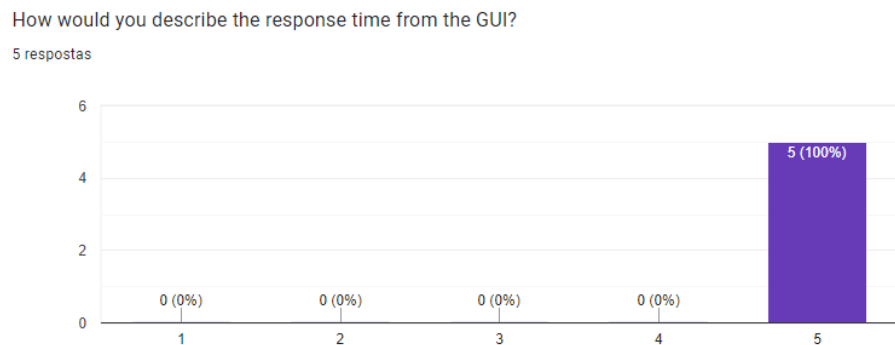


Figure 6.5: Fifth question.

be displayed at the beginning of the campaigns menu, some units were missing in the graphs and in the construction of the graph the available parameters were not organised and did not have a concrete name which made it confusing.

Figure 6.3 and Figure 6.5 allows to verify two of the non-functional requirements in Section 4.4.2 regarding the consistency of the organization of the various components and the speed of response by the graphical interface. Figure 6.3 received a good rating overall and Figure 6.5 received an excellent rating. The final two questions of the questionnaire allowed participants to express positive aspects and what had not worked well. The positive aspects highlighted by the participants are:

- Well organised and clean sequence of steps;
- A lot of information to configure and choose that is well organised;
- The use of graphics is professional and practical;
- Graphics were responsive and beautiful;
- Graphical interface appealing and simple to use;
- The campaign creation flow is very intuitive and gives a sense of progress;
- The amount of data made available by the tool to analyse the results;
- Very fast application performance with very little latency.

The negative aspects are listed below:

- Lack of help for less experienced users;
- A bug in the transformers' components;
- Complicated to know which page the user is on by the breadcrumb displayed;
- The organisation of the data to be seen on the graph, as there is little descriptive text about each input parameter;
- The data in the selects to build the graph should be arranged alphabetically;
- Clicking on the logo in the top corner should take the users to the home-page;
- The graphs are missing units;
- On the home screen the most recent runs should appear first, i.e. in descending date order.

6.2.3 Test conclusions

Overall, participants were able to perform the tasks efficiently, with some exceptions, e.g. task T7 and T15 where some features were not perceptible. The organisation of the information both in the menu and on the graph creation page should be improved as well as other simple details.

Despite the negative aspects mentioned, the participants were able to understand the functionality and purpose of the framework, as shown in Figure 6.1 all participants thought that this type of framework with graphical interface is useful to simplify the effort applied by the users. The graphical interface managed to fulfil its essential role in improving many of the weakest aspects of the previous version, and simple usability aspects that ended up having an impact on user performance should be improved.

Chapter 7

Towards accelerating fault injection using failure models

Fault injection using fault models has been widely used for evaluating the dependability of systems and to validate fault tolerance mechanisms. However, despite being effective, it is a slow process because many faults do not have any effect (i.e., do not cause any visible failure in the target system). Fault injection using failure models may, hypothetically, be able of reproducing the same results, with similar levels of accuracy and representativeness, but at a fraction of the time and cost. Although failure models have been used before, based on state of the art, no study has verified whether the produced results are representative nor that failure models bring a speed and cost improvement. Failure models can reproduce both hardware failures, simulating crashes, disk failures, and software failures, for instance, corrupting return values. This research can start a path of possibilities related to accelerating fault injection through the application of failure models.

As the aim of this chapter it is intended to perform fault injection using fault and failure models and compare the results obtained in order to verify and validate the points mentioned above. ucXception 2.0 being a fault injection framework and one of the focuses of the first objectives of this thesis is an advantage to use it to execute campaigns and collect data through probes and parsers. The advantages are: the fault injection framework is already developed, to which one has access, it contains fault injection tools using fault models, its usability is better than its previous version making the framework simpler to use and its extensibility to integrate a new fault injection tool that uses a failure model and a new parser.

For the experiments is used an experimental setup representing a cloud deployment, the Openstack, as the target system. OpenStack is a cloud operating system in which everything is managed and provided through APIs with common authentication mechanisms [47]. Openstack is divided into several services to allow users to use the components according to their need, such as, compute, storage, networking, orchestration, shared services, among others.

7.1 Methodology

In this section will be presented in detail each step that led to a construction of an experimental scenario that allowed collecting results for an analysis regarding the use of two different fault injection models. It begins by presenting the setup configured for the experiment (Section 7.1.1). In the following, the workload used to test the system is exposed (Section 7.1.2). Subsequently, the models used in this experiment are discussed and presented in detail (Section 7.1.3). The last section, (Section 7.1.4), depicts how the output of the workload execution was processed.

7.1.1 Setup

In order to run the experiments a physical setup was configured. Its specifications in terms of hardware and software are given in Table 7.1. The ucXception 2.0 was installed on the machine manually, thus without using any containerization technology. The framework was configured on the machine without the support of containerization, because the containerization step was not yet fully functional.

Component	Description
Operating system	Linux 4.14.89
Hypervisor	Xen 4.11.1
CPU	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
CPU(s)	4
Thread(s) per core	2
RAM	16 GB
Disk1	1T*7200 RPM
Disk2	10T*7200 RPM

Table 7.1: Experimental setup specification

Openstack contains a plethora of services that are optional. In our setup, the 3 most common Openstack services were configured. One service (Nova) supports the creation of virtual machines and provides an API and tools for managing the resources of the cloud. Another service (Neutron) provides "network as a service" between interface devices managed by other Openstack services, such as Nova. The hypervisor used in Neutron to host the virtual machines was KVM/Qemu 4.2.1. Finally, the third installed service (Cinder) is a block storage service and is designed to present storage resources to end users that can be consumed by Nova.

Figure 7.1 illustrates how the setup is configured and how fault collection and injection is performed. Two pre-probes were configured to collect metrics regarding the three configured services. For Nova, the *Logs probe* was used to extract logs from the target system, in this case Openstack, and the *Ping probe* was configured to perform pings on the three services to monitor the various systems. ucXception 2.0 executes the workload and launch the probes for each service installed. While the workload is being executed, the framework manages the fault

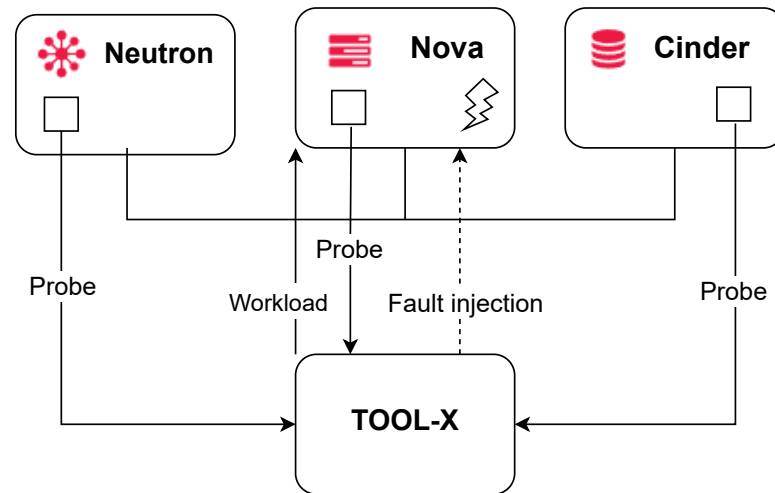


Figure 7.1: Experimental Setup.

injection process.

7.1.2 Workload

A workload automatically generates work tasks in a system. Its use is essential when testing the behaviour of a system, whether through fault injection, stress tests, among others. The workload used for the experiments was developed by a PhD student with some necessary adjustments so that it would be easier to build the parser that analyses the output generated by it.

The workload consists of several types of requests made to Openstack, which represent some of the most common operations that a system administrator might perform, such as listing, creating and deleting flavors or instances. The workload follows a sequential flow, starting by listing the flavors and instances of the moment, then it makes requests to create new elements and later it lists them again, finally it deletes the elements that were created and finishes listing the flavors and instances. The workload is simplistic and can be extended in the future to perform other operations. The workload takes on average 130 seconds to finish. In total performs 11 operations, although some are repeated more than once. The operations are:

1. List all flavors (i.e., the resource configurations that can be used by the virtual machines)
2. List all instances (i.e., virtual machines)
3. Create a new flavor with a certain configuration
4. Create a new instance using the previously defined flavor
5. Delete a flavor
6. Delete an instance

7.1.3 Injection process

Regarding the injection process, it is defined what, where and when to inject based on the setup and the goals of the study. The objective was to use fault and failure models to emulate transient hardware faults affecting a process of Openstack. Software and other kinds of faults were not considered.

Regarding the fault model, the single bit-flip fault model was used. This model emulates soft errors affecting the CPU register file directly or other CPU components (buses, ALU, FPU, etc.) indirectly. Both the bit and register are chose randomly following an uniform distribution, once per run.

For the failure model, it injects crashes of a process, which is a common failure mode obtained in fault injection experiments where a single bit-flip is injected in a random CPU register. Other failure models can be evaluated, however initially this mode was chosen because it is simple to implement and emulates a large portion of failures.

A new tool was developed which randomly chooses and kills (by sending a SIGKILL) a random process of nova-api, which is the service that receives the operation requests and passes them on to the correct service to be handled. Due to the extensibility of framework it was possible to add the new failure model injection tool that was developed later to the framework.

One of the various Openstack-related services of the Nova VM was chosen to be the target, as Nova is the central element of the setup. In the future it is planned to conduct similar experiments in the other services of Nova and of the other Openstack components. As the workload takes about 130 seconds to execute, the injection time was set to the range between]10,100[seconds, chosen randomly. The first 10 seconds correspond to the warmup and the last 30 seconds are the cooldown, which allow the injected faults to manifest and cause failures. In total, running a campaign of about 100 iterations takes about 34 hours, so 1000 runs takes about 2 weeks.

7.1.4 Failure detection

The expected output from running the workload includes information about the return code of each operation (which indicates whether the operation executed successfully or not), the duration of its execution and, for some operations, the output produced by the operation. These parameters allow a detailed analysis to be made of the output during the experiments, with the aim of assessing whether the result after fault injection remains as expected. For automatically performing the data processing, a parser was developed and integrated into the ucXception 2.0. For each operation of the workload, the parser processes the workload output and writes the following data to the Comma-Separated Values (CSV) file: Correct or incorrect output and the respective output size, status code, total time taken to perform the operation.

Another measure that must be considered, but that is not treated by the parser, is

the watchdog parameter that is set at the beginning of the campaign configuration. As the execution of the workload takes about 130 seconds, a watchdog of 200 seconds was defined. This time is longer than the execution of the workload so that the workload has time to finish by itself. If the workload takes longer, then the watchdog terminates the process as to avoid waiting for a possibly hanged process.

These metrics allow the results to be classified in two ways: 1) No Effect (the workload executes seemingly without problem), and 2) Failed (at least one operation returned a status code different from expected, thus signaling an unsuccessful operation). Silent data corruption, despite important, was not included because no occurrence was detected.

7.2 Results

A total of 2000 runs were executed, equally distributed between injection using fault and failure models. Applying the classification scale resulted in 94% of No Effect for the failure model and 98% for the fault model. Therefore there were 6% of failed runs when using the failure model and 2% with respect to the fault model.

Figure 7.2 shows a comparison between the amount of operations that failed in a single run for fault and failure models. It should be noted that the Y-axis is zoomed in between 0 and 4.5%. An analysis of the results concludes that not only failure models cause more failures, the generated failures also affected more operations, on average.

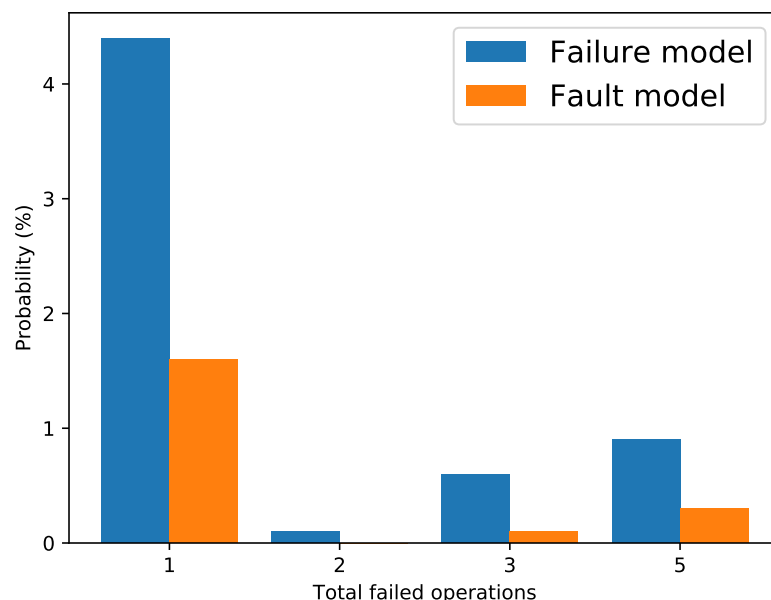


Figure 7.2: Failed operations per run for both models.

Figure 7.3 shows the probability of an error occurring in each of the various operations of the workload. Once again the Y-axis is zoomed in. The graph shows that

operation T7 has a higher failure probability when using failure models which can be explained by being a more sensitive operation or by the moment of injection. For the failure model it can be noted that from T5 to T8 have practically the same probability which also indicates that the failures are more concentrated in those specific tasks. The two first and last operations did not experience any failure likely due to the warmup and cooldown periods.

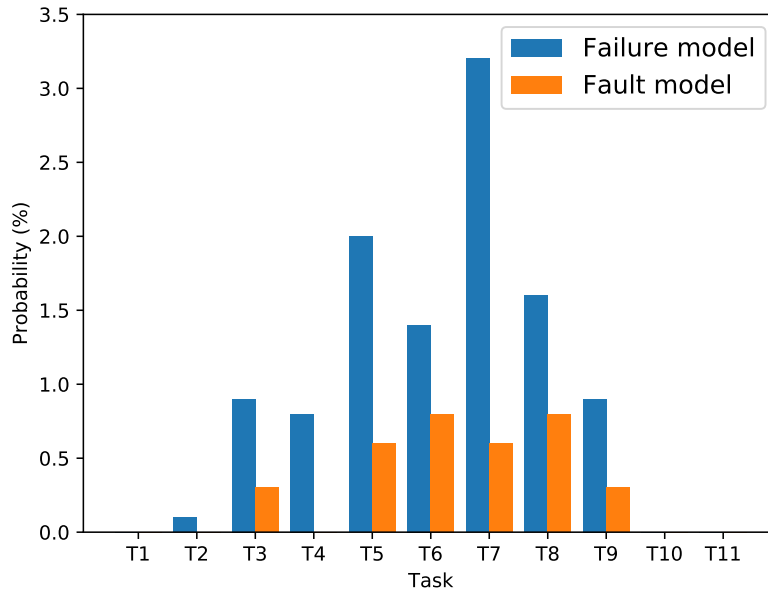


Figure 7.3: Failures per operation for both models.

To measure how the failure probability evolves when using failure models, Figure 7.4 plots the distance between the average failure probability obtained in each run when using failure models against the final failure probability obtained using fault models (i.e., 2%). For a more visual explanation the following formulas can be observed.

$$FP_{failure_j} = 100 * \frac{Totalerrors_j}{j}$$

$$FP = \sum_{j=1}^{N_{runs}} FP_{failure_j} - 2\%$$

It can be seen that the rise over the runs is very steep which indicates that the probability of failure through the failure models over the runs manages to be higher than the probability of failure at the final failure probability using fault models.

Figure 7.5 shows the difference between failure probabilities across runs for each model. While for the failure models one can observe the sharp rise mentioned earlier, for the fault models it remains practically constant which indicates that the number of runs that fail does not change significantly over the runs.

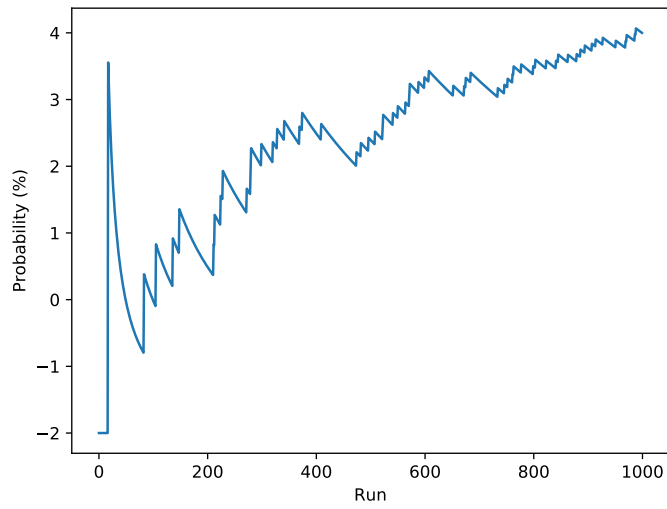


Figure 7.4: Distance between failure model and oracle.

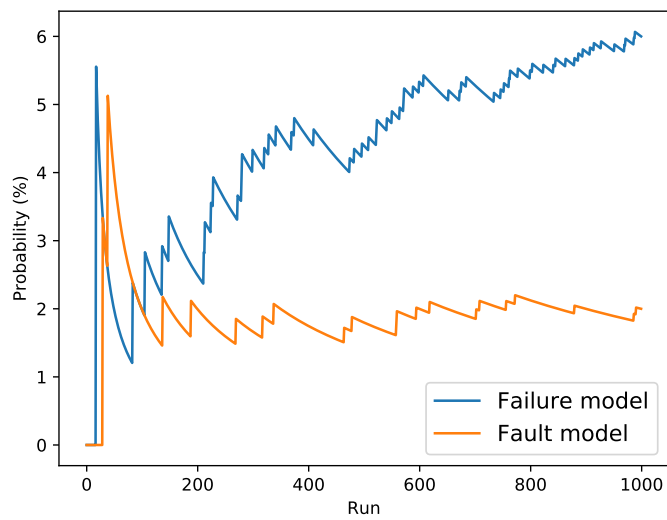


Figure 7.5: Distance between failure model and oracle.

7.3 Analysis & Limitations

The first analysis to emerge from the results is that the use of failure models produces more failures than the use of fault models, thus accelerating this type of experimental campaign. As can be observed in Figure 7.5 for a low number of runs the percentage of failure starts to be higher in the failure model, growing continuously, while the fault model stabilises.

Another analysis is that the use of failure models also results in failures that differ from those generated by injecting faults. In Figure 7.3, it can be seen that failure distributions do not conform to the same pattern across different operations. For example, failure models caused failures that strongly affected operation T7 whereas fault models did not. Another example is in operation T4, which experienced failures when using failure models but never when using fault models.

Despite the progress made in comparing fault and failure models, the present study has several limitations that will be addressed in future work. First, this experiment focuses on a specific setup, so the analysis cannot be extended to other types of setups. The workload also imposes limited demands on the system's resources because it involves only a small but frequently used set of operations. As part of future experiments, other workloads should be considered. Different faultloads, such as different fault models, can also be considered. Lastly, the total number of experiments run is still relatively low, due not only to problems encountered but also to lack of time.

Chapter 8

Conclusion

The ucXception framework allows to conduct fault injection campaigns easily, with the goal of simplifying this procedure and providing data for analysis in an organized way, however the poor usability of the framework becomes a problem when users with less experience in the area intend to use the framework. Furthermore, ucXception is difficult to configure, as it requires more advanced knowledge which users may not have.

To solve this problem, as first objective, ucXception 2.0 was developed, which includes a graphical user interface, thus simplifying the use of the framework. In order to fulfil the first objective, requirements were initially raised through the use of user stories, mockups, functional and non-functional requirements allowing an analysis of the most important requirements. An architecture was planned and an analysis of the technologies to be used was made. After the planning of ucXception, the second phase of this objective involved the development of the various functionalities raised. Two modules were developed, Frontend, represented by the implementation of the graphical interface presented to the users, and the Backend, which was the brain, the central component, of the whole framework. Furthermore, to make ucXception 2.0 accessible to multiple users and to configure it easily and quickly on the users' system, a containerisation technology was applied, Docker, to allow further extensibility and modifiability of the framework. The images created through Docker technology were later published in a public repository, Docker Hub.

During development two problems were encountered which had a major impact on the realisation of the new version of the framework. The first major difficulty is related to being able to translate the various campaigns and components of the framework to the graphical interface and backend in order to specify which parameters are needed to build them. The second issue is related to the Docker container being unable to access the target remote system. A solution has been found for both problems, but it has caused scheduling to be delayed.

The framework was tested for robustness to ensure that the central component of the framework would be as bug-free as possible, so it was possible to detect several errors that affect the performance of the framework. Moreover, the usability of the graphical interface was tested, thus being able to detect several problems.

Despite the problems, the general opinion of the participants was mostly positive as the execution of the proposed tasks, which indicates that the application fulfills its purpose.

For future work concerning the ucXception 2.0 framework, it is intended to implement all the functional requirements that were not developed, to polish/improve the code for future developers, to develop a queue for the execution of campaigns so that the system is not overloaded and also to make the framework save the results after each execution of a campaign. In addition, the most confusing aspects of the framework's usability pointed out in the usability tests should be corrected.

Another stipulated objective was to develop a failure model that could accelerate the process of fault injection into systems and be as representative as the fault models, for this purpose experiments were performed with the two models and later the results were compared and analysed. To do so, an experimental setup was configured using three Openstack services, a workload was defined and a parser was developed to collect statistics of the workload output, such as the status code of the operations, the time elapsed and verification of the output of each operation.

Previously, a study of the injection process was carried out, by which the type of faults and failures to be injected was chosen. It was used an existing fault model of the framework that allowed to inject single bit-flip faults. The failure model created allowed to crash a randomly chosen process of the nova-api service, was later added to the framework.

The results confirm that failure models can produce failures more frequently than fault injection, however the resulting failures may differ from those that occur when fault models are used. It is important to perform a similar fault injection using both models in a more vast number of operations, setup and number of runs in order to understand if the results are indeed representative. Further research is needed before a strong conclusion can be taken regarding this matter, which is planned as future work.

As planned, an article was written, as shown in Appendix C, which in a first phase presents in detail the framework developed, as architecture, reference examples, and its benefits. The second part of the article presents the research component carried out. It addresses the entire methodology of the applied process, such as setup, workload, among others, and presents the results between the failure and fault models and a respective analysis. Initially, only the writing of the article was planned, however with the delay of the planning, an opportunity was found to submitted to the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC) conference. As the event has been postponed, we are currently awaiting a response.

References

- [1] ISO 9241-11. International organization of standardization. guidance on usability. 1998., (Accessed May 20, 2022).
- [2] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Goofi: generic object-oriented fault injection tool. In *2001 International Conference on Dependable Systems and Networks*, pages 83–88, 2001.
- [3] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, 1990.
- [4] Avi. 8 popular python frameworks to build api. <https://geekflare.com/python-frameworks-for-apis/>, (Accessed November 30, 2021).
- [5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.
- [6] Raul Barbosa, Johan Karlsson, Henrique Madeira, and Marco Vieira. *Fault injection*, pages 263–281. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [7] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. Fault injection experiments using fiat. *IEEE Transactions on Computers*, 39(4):575–582, 1990.
- [8] WINI BHALLA. How to build apis in python: 8 popular frameworks. <https://www.makeuseof.com/build-api-python-popular-frameworks/>, (Accessed November 30, 2021).
- [9] Ricardo Camacho. Robustness testing: What is it & how to deliver reliable software systems with test automation. <https://www.parasoft.com/blog/what-is-robustness-testing/>, (Accessed August 17, 2022).
- [10] J. Carreira, H. Madeira, and J.G. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, 1998.
- [11] Frederico Manuel Duarte Cerveira. *Evaluating and improving cloud computing dependability*. PhD thesis, 00500:: Universidade de Coimbra, 2021.
- [12] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. In *Proceedings of Annual Symposium on Fault Tolerant Computing*, pages 304–313, 1996.

- [13] J. Christmansson, M. Hiller, and M. Rimen. An experimental comparison of fault and error injection. In *Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No.98TB100257)*, pages 369–378, 1998.
- [14] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. Profipy: Programmable software fault injection as-a-service. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 364–372, 2020.
- [15] Shaumik Daityari. Angular vs react vs vue: Which framework to choose. <https://www.codeinwp.com/blog/angular-vs-vue-vs-react/>, (Accessed November 29, 2021).
- [16] Django. The web framework for perfectionists with deadlines. <https://www.djangoproject.com/>, (Accessed November 30, 2021).
- [17] Docker. Docker overview. <https://docs.docker.com/get-started/overview/>, (Accessed December 6, 2021).
- [18] 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.
- [19] Falcon. Falcon is a minimalist wsgi library. <https://falcon.readthedocs.io/en/stable/>, (Accessed November 30, 2021).
- [20] N.E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.
- [21] Flask-RESTful. Flask-restful provides an extension to flask. <https://flask-restful.readthedocs.io/en/latest/>, (Accessed November 30, 2021).
- [22] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Edfi: A dependable fault injection tool for dependability benchmarking experiments. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, pages 31–40, 2013.
- [23] Michael Grottke and Kishor S Trivedi. A classification of software faults. *Journal of Reliability Engineering Association of Japan*, 27(7):425–438, 2005.
- [24] Nielsen Norman Group. Response times: The 3 important limits. <https://www.nngroup.com/articles/response-times-3-important-limits/>, (Accessed February 11, 2022).
- [25] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.
- [26] inVerita. Vue vs react vs angular: What framework would you choose? <https://medium.com/swlh/vue-vs-react-vs-angular-what-framework-would-you-choose-5d77a3680b0d>, (Accessed November 29, 2021).

-
- [27] Niklas Johansson and Anton Löfgren. Designing for extensibility: An action research study of maximizing extensibility by means of design principles. B.S. thesis, University of Gothenburg, 2009.
- [28] Pallavi Joshi, Haryadi Gunawi, and Koushik Sen. Prefail: a programmable tool for multiple-failure injection. In *OOPSLA '11*, volume 46, pages 171–188, 10 2011.
- [29] Jumpgrowth. Title of citation. <https://jumpgrowth.com/top-10-web-development-frameworks/>, (Accessed November 29, 2021).
- [30] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. Ferrari: a flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2):248–260, 1995.
- [31] W.-I. Kao, R.K. Iyer, and D. Tang. Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, 1993.
- [32] Dawid Karczewski. What are the best frontend frameworks to use in 2021? <https://www.ideamotive.co/blog/best-frontend-frameworks>, (Accessed November 29, 2021).
- [33] 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, 2014.
- [34] Jay A. Kreibich. *Using SQLite*. O'Reilly Media, Inc., 2010.
- [35] Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas, and Karthik Pat-tabiraman. Lfi: An intermediate code-level fault injection tool for hardware faults. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 11–16, 2015.
- [36] Lucidchart. What is an entity relationship diagram (erd)? <https://www.lucidchart.com/pages/er-diagrams>, (Accessed August 03, 2022).
- [37] H. Madeira, D. Costa, and M. Vieira. On the emulation of software faults by software fault injection. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 417–426, 2000.
- [38] H. Madeira and J. Durães. Emulation of software faults by educated mutations at machine-code level. In *Proceedings 13th International Symposium on Software Reliability Engineering*, page 329, Los Alamitos, CA, USA, nov 2002. IEEE Computer Society.
- [39] Paul D. Marinescu and George Candea. Lfi: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 379–388, 2009.

- [40] E. Martins, C.M.F. Rubira, and N.G.M. Leme. Jaca: a reflective fault injection tool based on patterns. In *Proceedings International Conference on Dependable Systems and Networks*, pages 483–487, 2002.
- [41] Rolando Martins, Rajeev Gandhi, Priya Narasimhan, Soila Pertet, António Casimiro, Diego Kreutz, and Paulo Veríssimo. Experiences with fault-injection in a byzantine fault-tolerant protocol. In *Acm/ifip/usenix international conference on distributed systems platforms and open distributed processing*, pages 41–61. Springer, 2013.
- [42] Kate Moran. Usability testing 101. <https://www.nngroup.com/articles/usability-testing-101/>, (Accessed August 19, 2022).
- [43] S.S. Mukherjee, J. Emer, and S.K. Reinhardt. The soft error problem: an architectural perspective. In *11th International Symposium on High-Performance Computer Architecture*, pages 243–247, 2005.
- [44] 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.
- [45] Jakob Nielsen. Why you only need to test with 5 users. <https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/>, (Accessed August 19, 2022).
- [46] Jakob Nielsen. Usability 101: Introduction to usability. <https://www.nngroup.com/articles/usability-101-introduction-to-usability/>, (Accessed May 20, 2022).
- [47] OPENSTACK. Software. <https://www.openstack.org/software/>, (Accessed February 18, 2022).
- [48] Thomas Ostrand and Elaine Weyuker. The distribution of faults in a large industrial software system. *ACM SIGSOFT Software Engineering Notes*, 27:55–64, 07 2002.
- [49] Pradeep Parthiban. 7 reasons why software testing is important. <https://www.indiumsoftware.com/blog/why-software-testing/>, (Accessed August 17, 2022).
- [50] Gonçalo Pereira, Raul Barbosa, and Henrique Madeira. Practical emulation of software defects in source code. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 130–140, 2016.
- [51] Matei Ripeanu. Software architecture in practice. <https://people.ece.ubc.ca/matei/EECE417/BASS/ch041ev1sec4.html>, (Accessed February 17, 2022).
- [52] Nicholas Samuel. What is sqlite? <https://www.sqlite.org/index.html>, (Accessed December 1, 2021).
- [53] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. Fail*: An open and versatile fault-injection framework for the assessment of software-implemented hardware

- fault tolerance. In *2015 11th European Dependable Computing Conference (EDCC)*, pages 245–255, 2015.
- [54] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Ysskin, J. Kownacki, J. Barton, R. Dancy, A. Robinson, and T. Lin. Fiat - fault injection based automated testing environment. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.*, pages 394–, 1995.
- [55] Lakshay Sharma. Why is testing necessary? <https://www.toolsqa.com/software-testing/istqb/why-is-testing-necessary/>, (Accessed August 17, 2022).
- [56] Vadym Shliachkov. Sample size for usability study. part 1. about nielsen and probability. <https://uxplanet.org/sample-size-for-usability-study-part-1-about-nielsen-and-probability-effectiveness/> (Accessed August 19, 2022).
- [57] Nuno Silva, Ricardo Barbosa, João Carlos Cunha, and Marco Vieira. A view on the past and future of fault injection. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–2, 2013.
- [58] 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, 2010.
- [59] Ningfang Song, Jiaomei Qin, Xiong Pan, and Yan Deng. Fault injection methodology and tools. In *Proceedings of 2011 International Conference on Electronics and Optoelectronics*, volume 1, pages V1–47–V1–50, 2011.
- [60] D.T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R.K. Iyer. Nf-tape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*, pages 91–100, 2000.
- [61] Swagger. Openapi specification. <https://swagger.io/specification/>, (Accessed August 17, 2022).
- [62] TechSmith. Usability testing basics. <http://webservices.itcs.umich.edu/drupal/wwwsig/sites/webservices.itcs.umich.edu.drupal.wwwsig/files/Usability-Testing-Basics.pdf>, (Accessed May 20, 2022).
- [63] Adarsh Tripathi. Best front end frameworks for web development of 2021: The complete guide. <https://medium.com/geekculture/best-front-end-frameworks-for-web-development-of-2021-the-complete-guide-ec3> (Accessed November 29, 2021).
- [64] Timothy Tsai and RaviShankar Iyer. Ftape-a fault injection tool to measure fault tolerance. In *10th Computing in Aerospace Conference*, page 1041, 1995.

- [65] usability.gov. Usability testing. <https://www.usability.gov/how-to-and-tools/methods/usability-testing.html>, (Accessed May 20, 2022).
- [66] E. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly "good" software can behave. *IEEE Software*, 14(4):73–83, 1997.
- [67] Vue.js. Comparison with other frameworks. <https://vuejs.org/v2/guide/comparison.html>, (Accessed November 29, 2021).
- [68] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *2009 IEEE International Conference on Software Maintenance*, pages 274–283, 2009.
- [69] Pingyu Zhang and Sebastian Elbaum. Amplifying tests to validate exception handling code. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 595–605, 2012.
- [70] 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.

Appendices

Appendix A

User stories

- **Authentication module**

- Register in framework

- * *US-1: As a user I want to* have an account **so that** I can have my experiences saved.

- * *Acceptance Criteria: Given that* I am a user **when** I register in the platform **then** the system will allow me to use the framework.

- Login into framework

- * *US-2: As a user I want to* login into my account **so that** I can access my campaigns and system functionalities.

- * *Acceptance Criteria: Given that* I am a user **when** I write my credentials and click to submit **then** the system will verify my credentials and according to its response will allow me or not to login.

- Reset password

- * *US-3: As a user I want to* reset password (e.g., possibly exposed, forgotten password) **so that** I can maintain it secure or regain access to the application.

- * *Acceptance Criteria: Given that* I am a user **when** I select to reset password **then** the system will allow me to write new password and save it.

- Logout

- * *US-4: As a user I want to* be able to logout from the application **so that** I can change account or keep my account secure while I am absent.

- * *Acceptance Criteria: Given that* I am a user **when** I click to logout **then** the system will clear all my authentication information and change the page.

- **Menu module**

- List campaigns

- * *US-5: As a user I want to* be able to view my campaign history **so that** I can review past campaigns and their results.
- * *Acceptance Criteria: Given that* I am a user **when** I click to view history **then** the system will display all my campaigns.
- Search/filter for a campaign
 - * *US-6: As a user I want to* be able to search for a specific campaign **so that** I can easily find it.
 - * *Acceptance Criteria: Given that* I am a user **when** I search for a campaign **then** the system will give campaigns according to that search.
- Order campaigns
 - * *US-7: As a user I want to* be able to order my campaigns by the date or name **so that** I can save time searching for an old campaign.
 - * *Acceptance Criteria: Given that* I am a user **when** I click to order my campaigns by the date or name **then** the system will display the campaigns ordered.
- Delete campaign
 - * *US-8: As a user I want to* be able to delete a campaign **so that** I can remove experiences that I do not need anymore.
 - * *Acceptance Criteria: Given that* I am a user **when** I click to remove a campaign **then** the system will delete the campaign.
- Cancel campaign
 - * *US-9: As a user I want to* be able to cancel a campaign **so that** I do not need to wait until the end.
 - * *Acceptance Criteria: Given that* I am a user **when** I click to cancel a campaign **then** the system will stop permanently the campaign.
- **Campaign setup module**
 - Choose fault injection tool
 - * *US-10: As a user I want to* choose which type of fault injection tool (e.g., SW, HW, Virtualized) to use **so that** I can focus in a specific fault type.
 - * *Acceptance Criteria: Given that* I am a user **when** I choose the fault injection tool **then** the system will change tool corresponding to the button pressed.
 - Configure fault injection tool
 - * *US-11: As a user I want to* configure the parameters of fault injection tool **so that** I can execute fault injection according to my analysis.
 - * *Acceptance Criteria: Given that* I am a user **when** I configure the fault injection tool **then** the system will launch the tool with that specific configuration.
 - Configure watchdog

-
- * *US-12: As a user I want to* be able to choose between to calculate automatically or to configure the time until the run stop **so that** the run does not extend too much or does not stay block.
 - * *Acceptance Criteria: Given that* I am a user **when** I insert the value or select the checkbox **then** the system will change the variable according to given value or calculate it automatically.
 - Upload program
 - * *US-13: As a user I want to* be able to upload source code or executable **so that** I can evaluate the uploaded program.
 - * *Acceptance Criteria: Given that* I am a user **when** I upload the file to the system and click to submit **then** the system will temporarily save the file and change page.
 - Create target/host
 - * *US-14: As a user I want to* be able to configure target system for fault injection **so that** I can run experiences where it is more practical.
 - * *Acceptance Criteria: Given that* I am a user **when** I configure the target host **then** the system will change is value and try to connect when launch.
 - Delete target/host
 - * *US-15: As a user I want to* be able to delete a configured target system **so that** it is not used during the execution of a campaign.
 - * *Acceptance Criteria: Given that* I am a user **when** I click to delete the target **then** the system will remove it from the campaign.
 - Create execution
 - * *US-16: As a user I want to* be able to configure a campaign execution **so that** I can define how many times the campaign will be executed.
 - * *Acceptance Criteria: Given that* I am a user **when** I configure the campaign execution **then** the system will change is value and associate it with the campaign.
 - Delete execution
 - * *US-17: As a user I want to* be able to delete a campaign execution **so that** it is not used in the execution of the campaign.
 - * *Acceptance Criteria: Given that* I am a user **when** I click to delete the execution **then** the system will remove it from the campaign.
 - Choose probes
 - * *US-18: As a user I want to* choose the probes to be used **so that** I can monitor the system **when** fault injection occurred.
 - * *Acceptance Criteria: Given that* I am a user **when** I select the probes **then** the system will launch the probes during the fault injection.
 - Choose transformers

- * *US-19: As a user I want to* be able to transform data in a more readable format **so that** I can understand the output from fault injection.
 - * *Acceptance Criteria: Given that* I am a user **when** I select transformers **then** the system will launch transformers after extracting data.
 - Choose validators
 - * *US-20: As a user I want to* choose a validator to use **so that** I can validate if the results are in an acceptance condition.
 - * *Acceptance Criteria: Given that* I am a user **when** I choose a validator or submit one **then** the system will launch the validator after fault injection.
 - Configure components
 - * *US-21: As a user I want to* configure the parameters of the components (input variables, probes, transformers, validators) used in campaign **so that** I can define the fault injection campaign according to my analysis.
 - * *Acceptance Criteria: Given that* I am a user **when** I insert the values in a form and click to submit **then** the system will change the components configuration before they are launched.
 - Delete component
 - * *US-22: As a user I want to* delete an undesirable component **so that** it is not executed during the campaign.
 - * *Acceptance Criteria: Given that* I am a user **when** I click to delete the component **then** the system will delete the component from the campaign.
 - Campaign Summary
 - * *US-23: As a user I want to* see a summary of the campaign **so that** I can get an overview of the campaign.
 - * *Acceptance Criteria: Given that* I am a user **when** I finish configuring the campaign **then** the system will display the campaign summary with all the configuration data.
 - Component Summary
 - * *US-24: As a user I want to* see a summary of all the components **so that** I can get an overview of the components.
 - * *Acceptance Criteria: Given that* I am a user **when** I finish configuring the components **then** the system will display the components summary with all the configuration data.
 - Execute campaign
 - * *US-25: As a user I want to* execute the campaign after configuring it **so that** I can get the results.
 - * *Acceptance Criteria: Given that* I am a user **when** I click to execute **then** the system will run the campaign.
- **Campaign menu module**

-
- Display data
 - * *US-26: As a user I want to* analyse the results in an easily understandable format **so that** it is simpler and more straightforward to analyse.
 - * *Acceptance Criteria: Given that* I am a user **when** I select a finish or ongoing campaign **then** the system will provide a consolidated report of results.
 - View statistics
 - * *US-27: As a user I want to* see some statistics **so that** I have a more detailed analysis of the campaign results.
 - * *Acceptance Criteria: Given that* I am a user **when** I choose to see the statistics data **then** the system will process the extracted data and display statistics relative to the runs and failure modes.
 - Build charts
 - * *US-28: As a user I want to* be able to build my own charts **so that** I can make an out of box analysis.
 - * *Acceptance Criteria: Given that* I am a user **when** I choose to create graphs and provide the axis values **then** the system will process the data and display the graph based on the axis values given.
 - View campaign raw data
 - * *US-29: As a user I want to* be able to see the raw data extracted from the campaign execution **so that** I can see the raw data without download it.
 - * *Acceptance Criteria: Given that* I am a user **when** I choose to see the raw data **then** the system will display the raw data as it is.
 - Download data
 - * *US-30: As a user I want to* be able to download/save the extracted and parsed data **so that** I can use it in reports or data analysis.
 - * *Acceptance Criteria: Given that* I am a user **when** I click on the Download button **then** the system will start downloading a CSV file.
 - Pause/resume campaign
 - * *US-31: As a user I want to* be able to pause and resume a campaign **so that** I can prioritize the campaigns according to circumstances (e.g., target system is down for maintenance).
 - * *Acceptance Criteria: Given that* I am a user **when** I click to pause or resume **then** the system will pause or resume the campaign.

Appendix B

Mockups

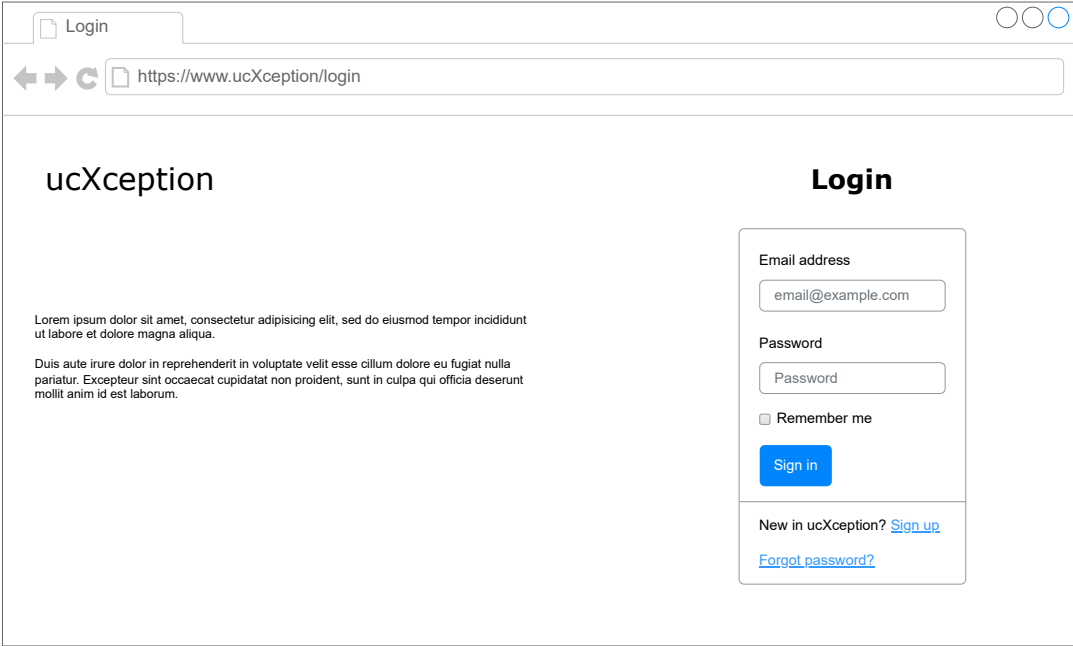


Figure B.1: Login page.

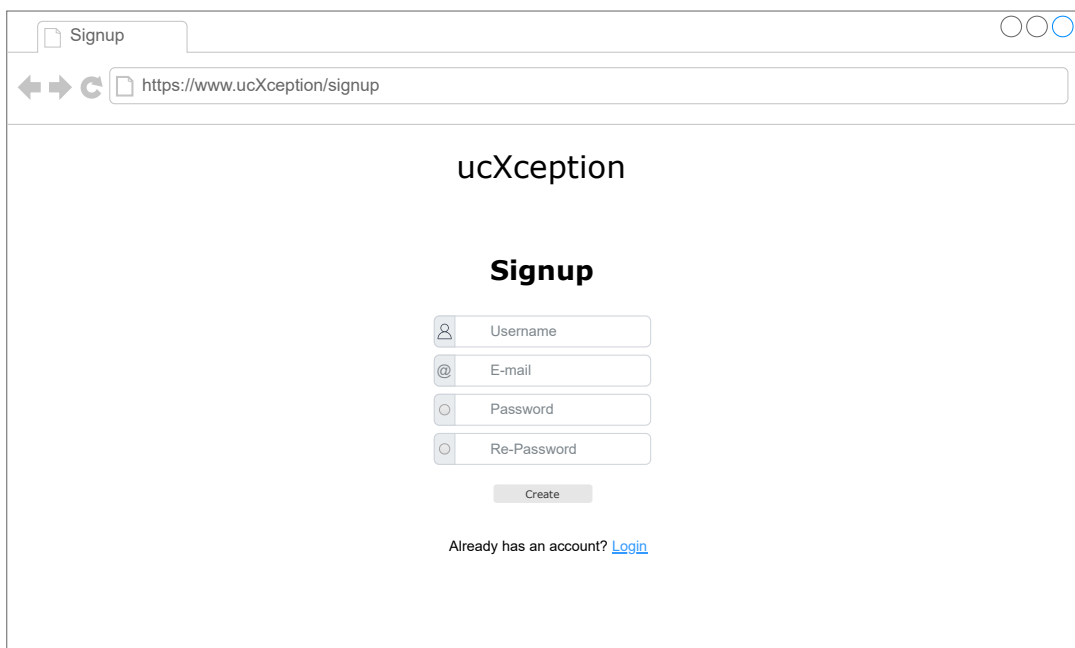


Figure B.2: Registration page.

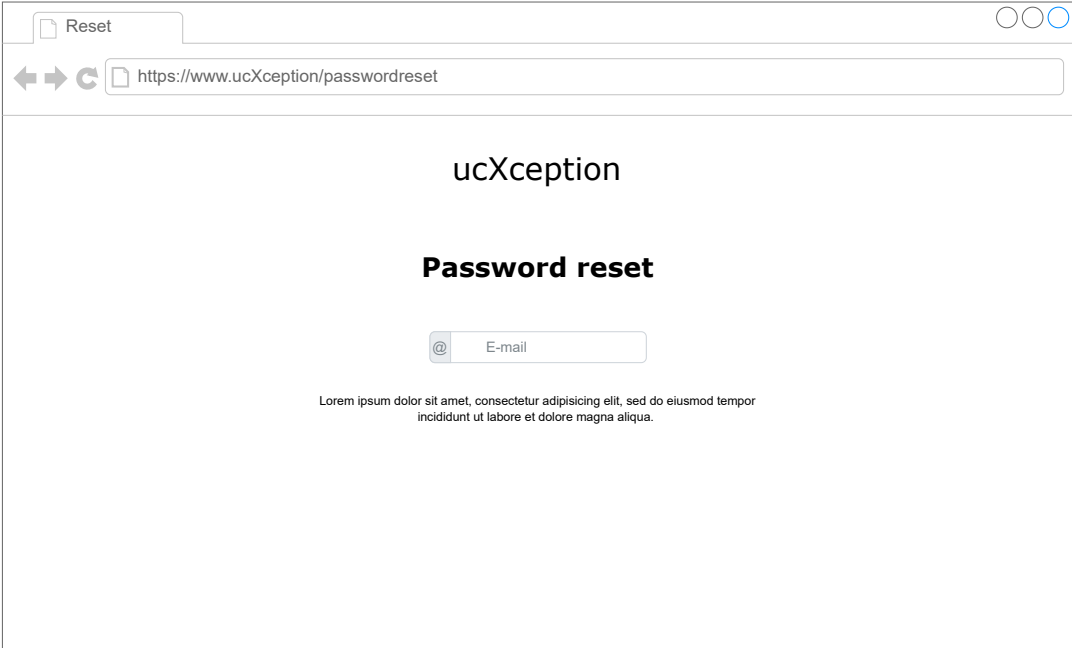


Figure B.3: Password reset page.

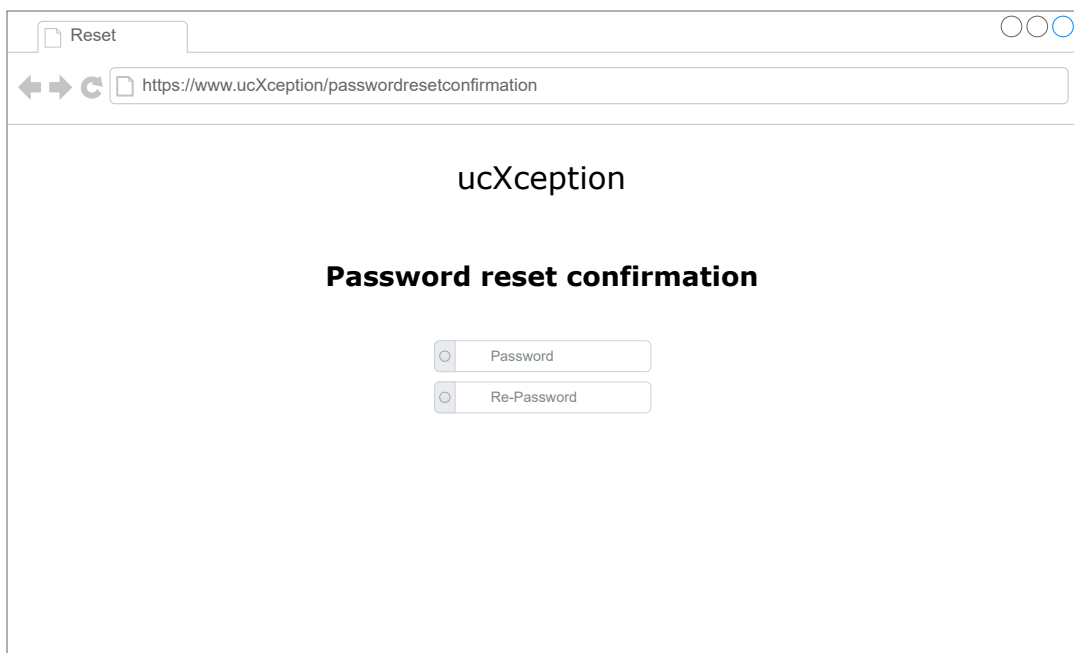


Figure B.4: Password reset confirmation page.

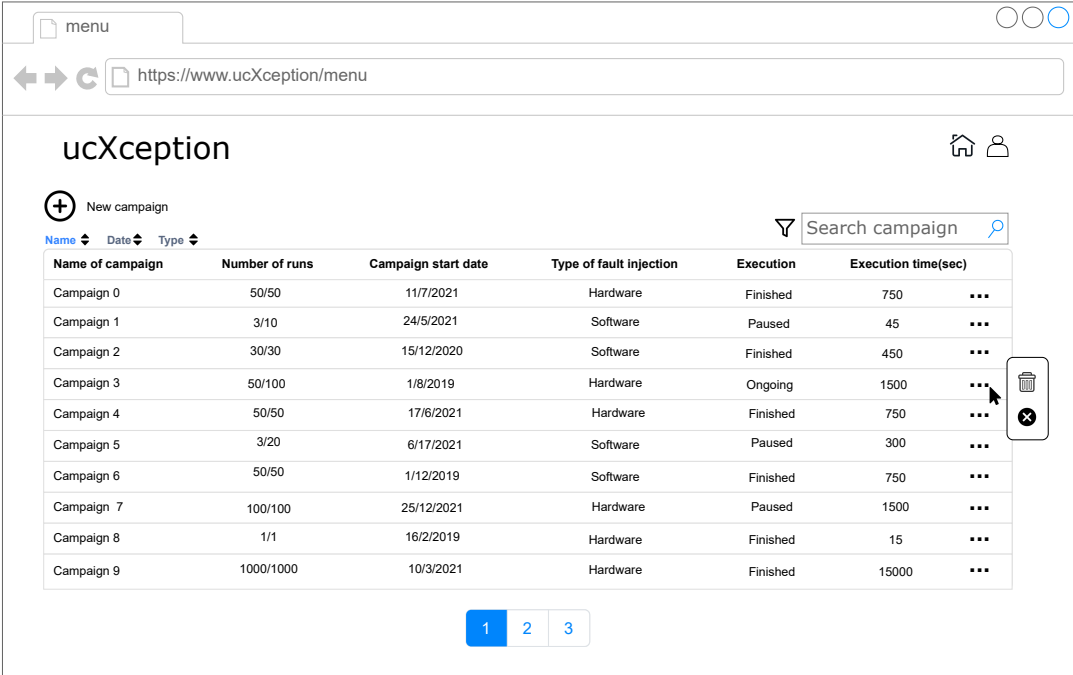


Figure B.5: Menu page.

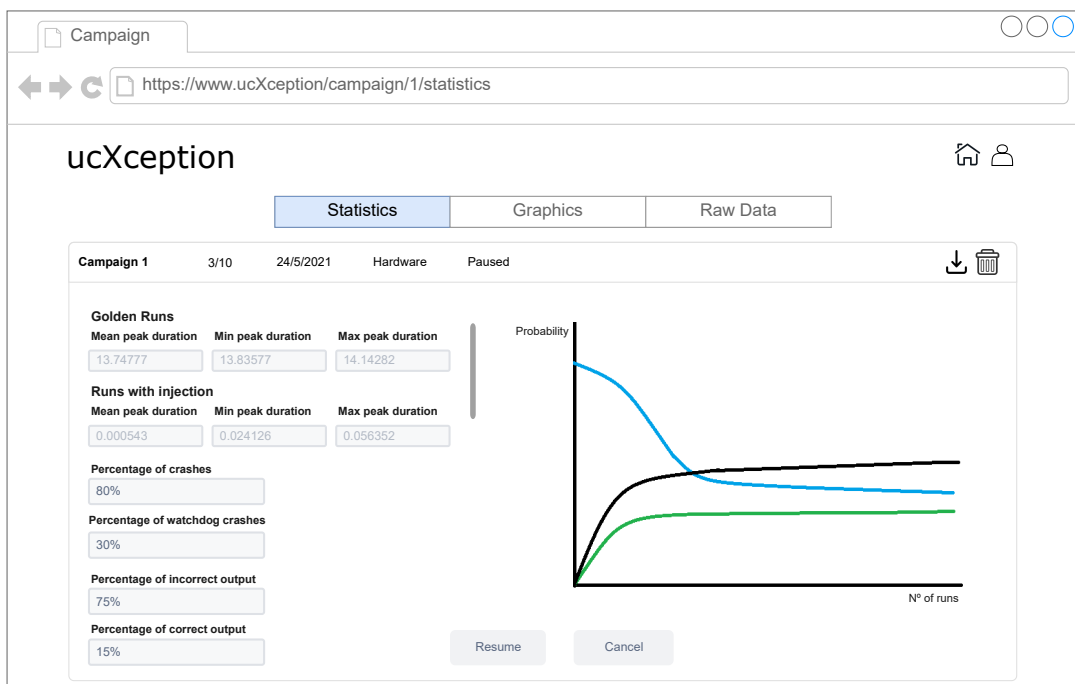


Figure B.6: Campaign statistics page.

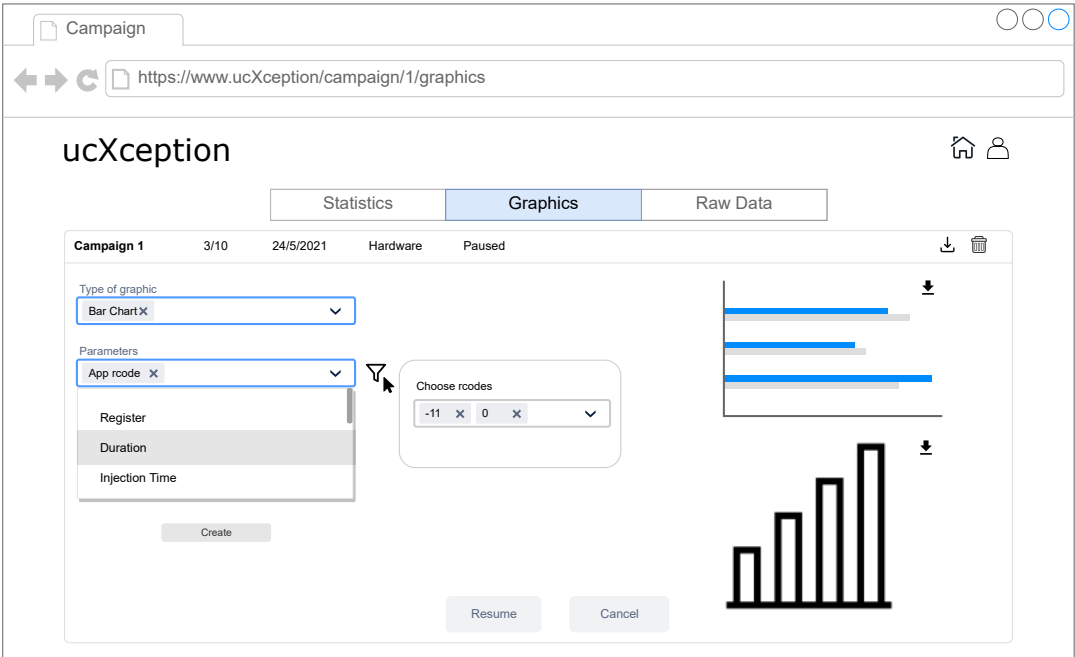


Figure B.7: Campaign graphics page.

The screenshot shows a web browser window with the address bar displaying `https://www.ucXception/campaign/1/rawdata`. The page title is "ucXception" and the browser tab is labeled "Campaign". The main content area features three tabs: "Statistics", "Graphics", and "Raw Data", with "Raw Data" being the active tab. Below the tabs, there is a header for "Experience 1" with details: "3/10", "24/5/2021", "Hardware", and "Paused". A table of raw data is displayed below the header, with columns for `app_rcode`, `bit`, `camp_name`, `dataextract_dur`, `dataextract_start`, `dataextract_stop`, `datavalidate_dur`, and `datavalidate_start`. The table contains three rows of data. Below the table is a horizontal scrollbar and two buttons: "Resume" and "Cancel".

app_rcode	bit	camp_name	dataextract_dur	dataextract_start	dataextract_stop	datavalidate_dur	datavalidate_start
-11	57	ai4eu_campaign	2.86E-06	1.64E-06	1.64E-06	0.000325	163242.342
1	56	ai4eu_campaign	3.81E-06	1.64E-06	1.64E-06	0.000246	163897.786
0	15	ai4eu_campaign	6.91E-06	1.64E-06	1.64E-06	0.000152	170013.925

Figure B.8: Campaign raw data page.

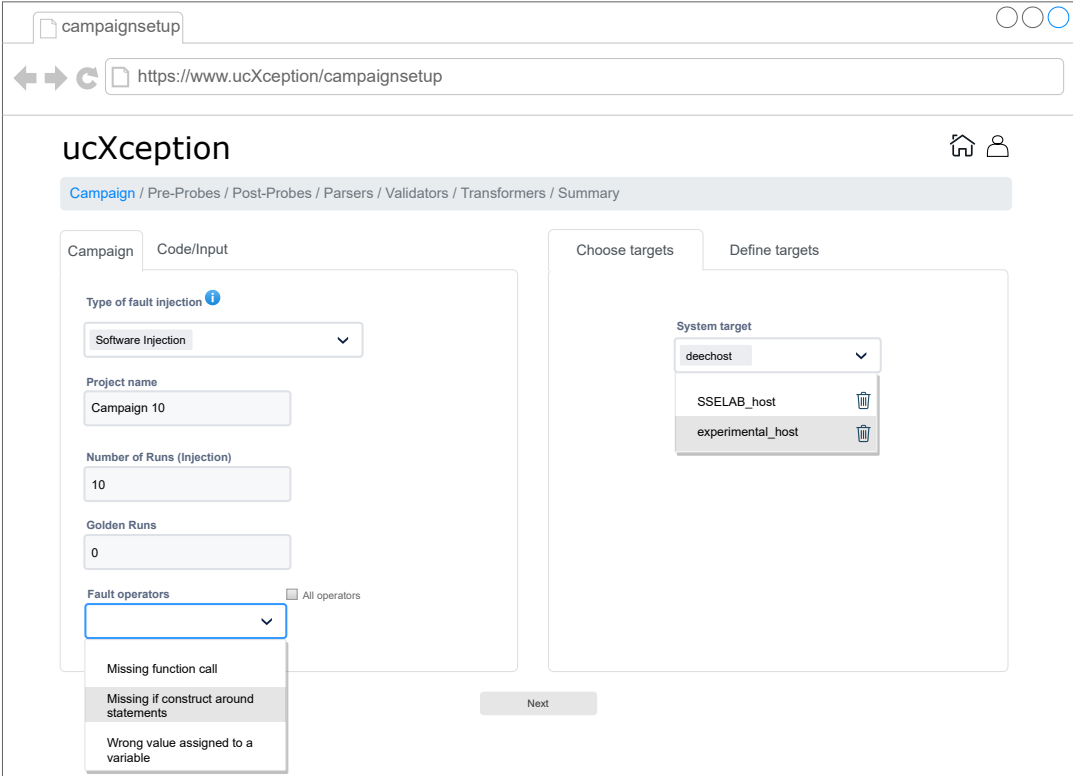


Figure B.9: Campaign configuration page.

The screenshot shows a web browser window with the address bar displaying `https://www.ucXception/campaignsetup`. The page title is "ucXception" and the breadcrumb trail is "Campaign / Pre-Probes / Post-Probes / Parsers / Validators / Transformers / Summary".

The main content area is divided into two sections: "Campaign" and "Code/Input".

Campaign Section:

- Application Input:** A text input field containing "3000".
- Whatchdog duration (ms):** A text input field containing "2000". A checkbox labeled "Calculate Automatically" is checked.
- Upload Code:** A dashed box containing a "Search File" button and a file upload icon.
- Upload scripts:** A dashed box containing a "Search File" button and a file upload icon.

Code/Input Section:

- Choose targets:** A tabbed interface with "Define targets" selected.
- System target:** Radio buttons for "Remote Host" (selected) and "Local Host".
- Domain:** A text input field containing "delhost".
- Username:** A text input field containing "root".
- Create host:** A button.

A "Next" button is located at the bottom center of the page.

Figure B.10: Campaign pre-probes configuration page.

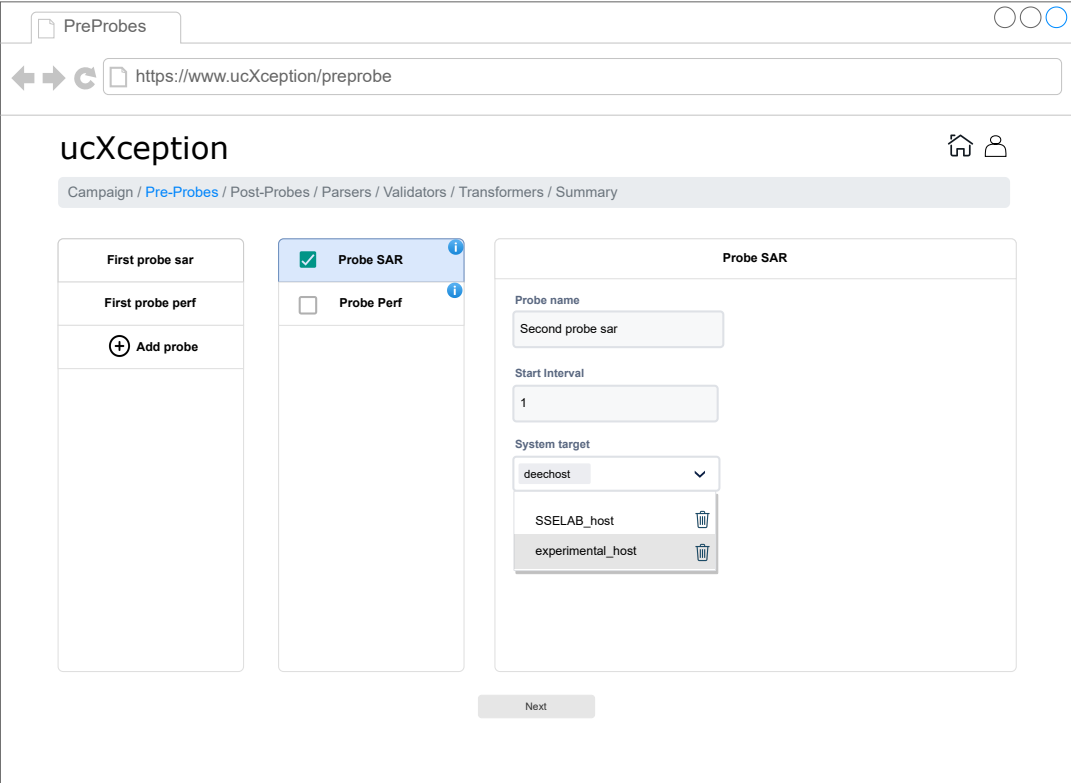


Figure B.11: Campaign post-probes configuration page.

The screenshot shows a web browser window with the URL `https://www.ucXception/postpobre`. The page title is "ucXception" and the breadcrumb navigation is "Campaign / Pre-Probes / Post-Probes / Parsers / Validators / Transformers / Summary".

The main content area is divided into three panels:

- Pidstat 1:** Contains a button "Second pidstat" and a button "Add probe".
- Pidstat:** Contains a checked checkbox and an information icon.
- Probe Pidstat:** Contains configuration fields:
 - Probe name: Second pidstat
 - Start Interval: 1
 - Process Name: Codeblocks
 - All process children: False
 - System target: deechost (dropdown menu)
 - SSELAB_host: [trash icon]
 - experimental_host: [trash icon]

A "Next" button is located at the bottom center of the configuration area.

Figure B.12: Campaign parsers configuration page.

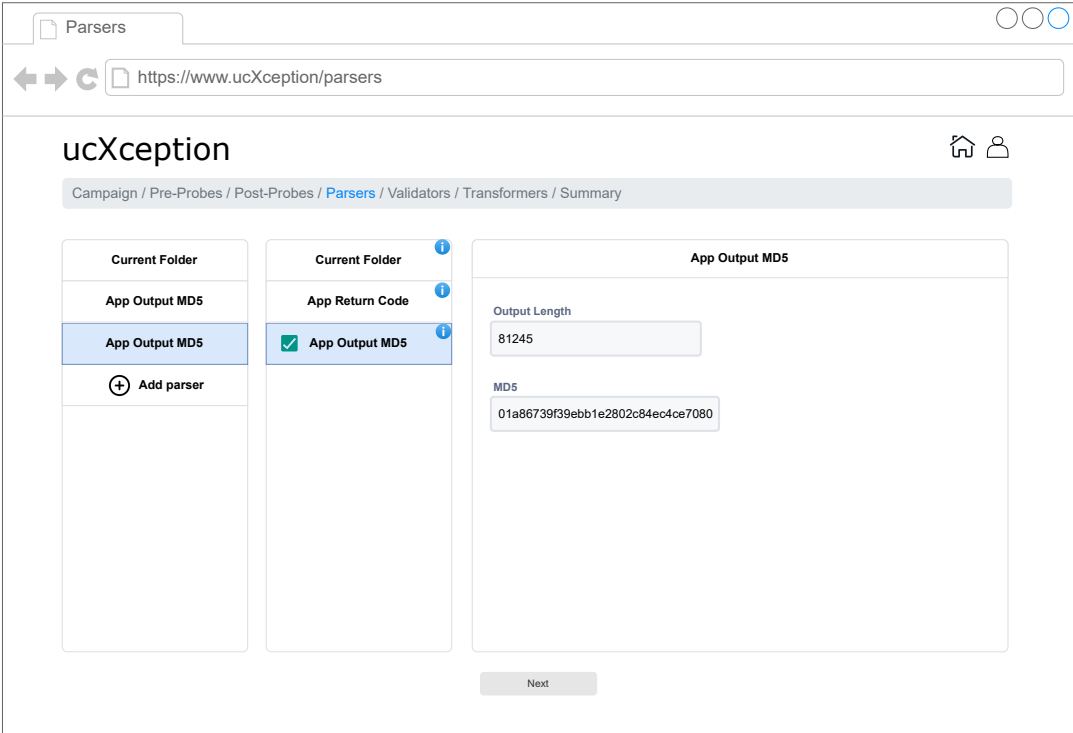


Figure B.13: Campaign validators configuration page.

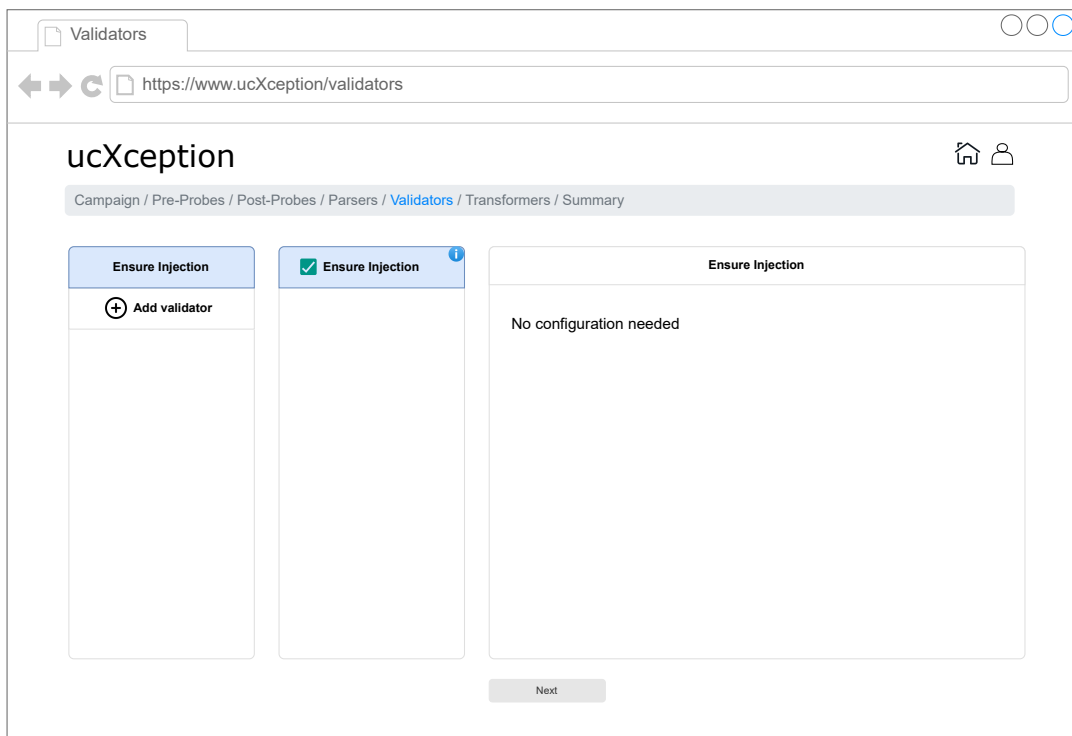


Figure B.14: Campaign transformers configuration page.

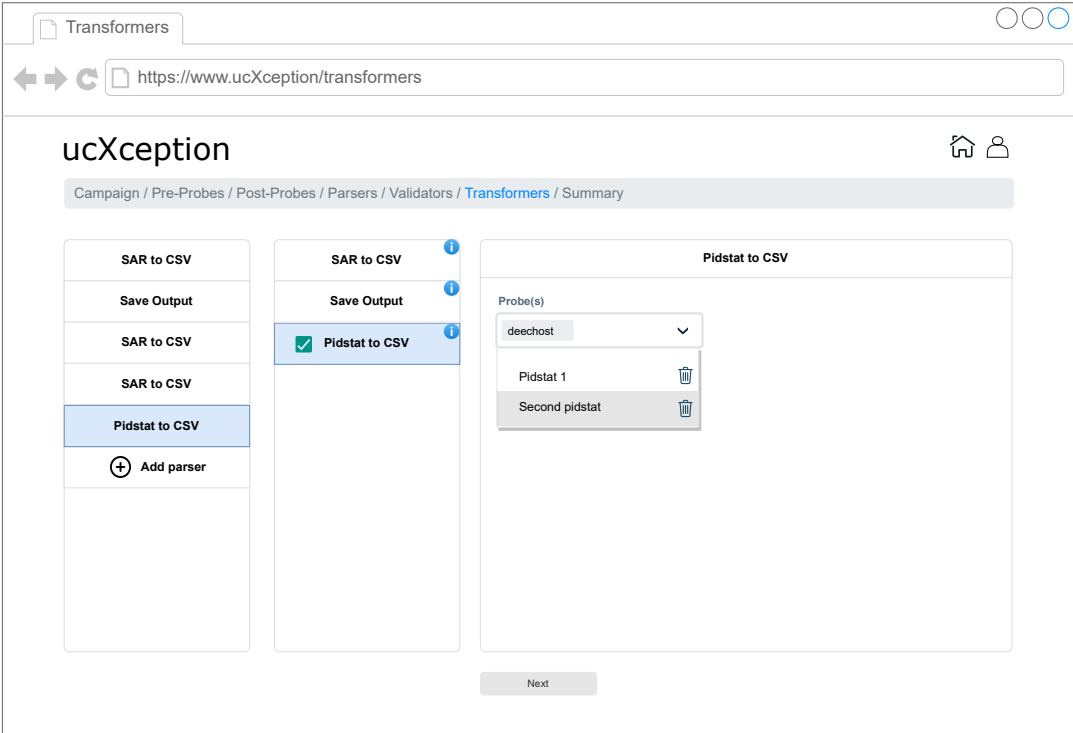


Figure B.15: Campaign configuration summary page.

The image shows a web browser window displaying the 'Summary' page of the ucXception application. The browser's address bar shows the URL 'https://www.ucXception/summary'. The page header includes the 'ucXception' logo and navigation icons. A breadcrumb trail reads 'Campaign / Pre-Probes / Post-Probes / Parsers / Validators / Transformers / Summary'. Below this, a tabbed interface is visible with 'Campaign' selected. The configuration area contains several input fields and buttons:

- Type of fault injection:** A dropdown menu with 'Software Injection' selected.
- Application Input:** A text input field containing '3000'.
- System target:** A dropdown menu with 'deechost' and 'SSELAB_host' as options.
- Project name:** A text input field containing 'Campaign 10'.
- Whatchdog duration (ms):** A text input field containing '2000'.
- Number of Runs (Injection):** A text input field containing '10'.
- Golden Runs:** A text input field containing '0'.
- Fault operators:** A row of five buttons labeled 'MIFS', 'MIA', 'MIEB', 'MFC', and 'MLAC'.

An 'Execute' button is located at the bottom center of the configuration area.

Figure B.16: Campaign summary configuration page.

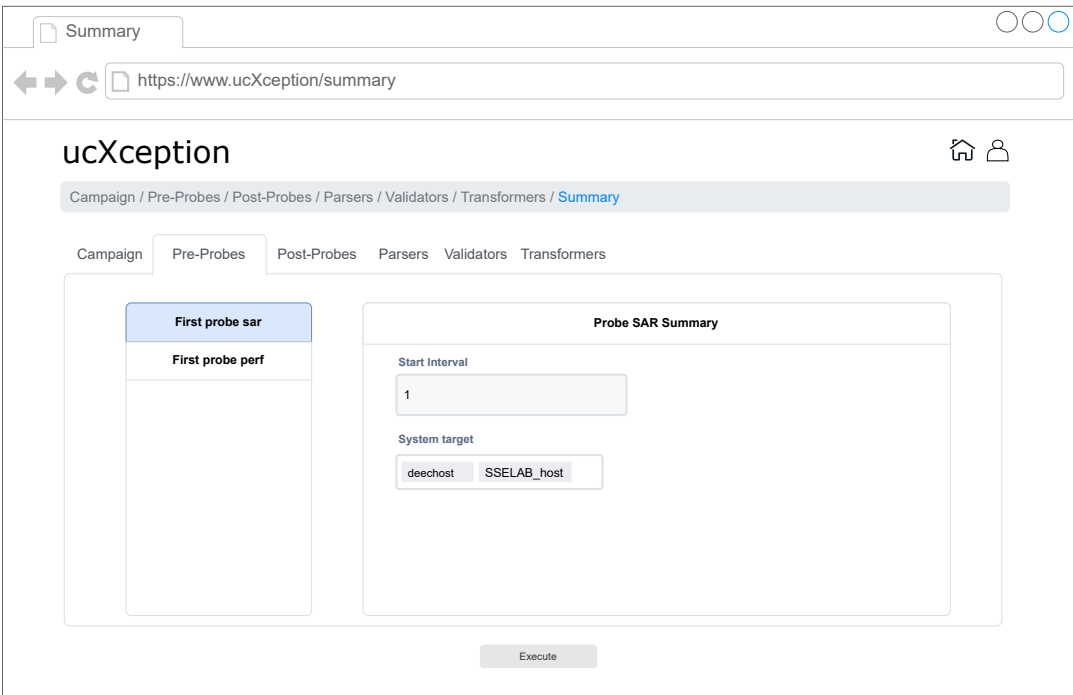


Figure B.17: Campaign pre-probes configuration summary page.

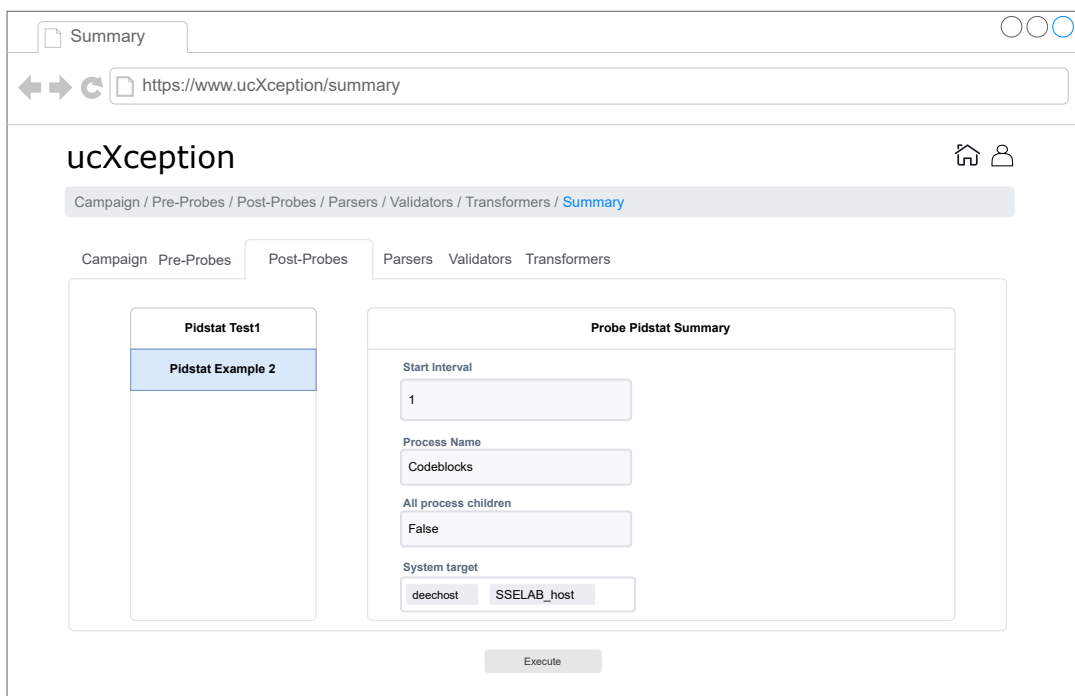


Figure B.18: Campaign post-probes configuration summary page.

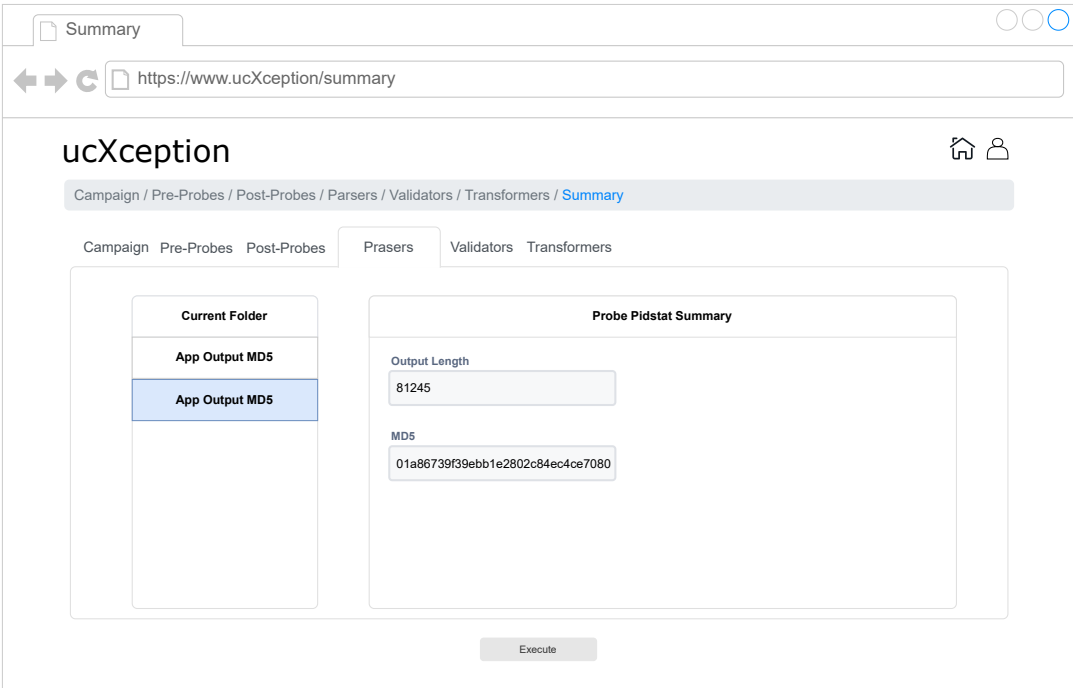


Figure B.19: Campaign parsers configuration summary page.

Appendix C

Article

TOOL-X: A framework for evaluating dependability of software systems

(Regular Paper)

Abstract—Fault injection is a well-established technique in the research community that consists of emulating faults using representative fault models, in order to obtain dependability-related data. Despite its potential, fault injection has been less widely adopted outside of academia, due to the expertise required to effectively conduct fault injection campaigns and to the lack of tools that can be easily adapted to different target systems. This paper presents TOOL-X, an easy-to-install, extendable, open source framework for conducting and orchestrating the entire lifecycle of fault injection campaigns without requiring expert knowledge and using a graphical interface. TOOL-X supports injection of software and hardware faults using realistic fault models and can be applied to a variety of target systems, including virtualized systems and complex cloud computing deployments. This brings fault injection to modern environments of cloud computing. In this paper, TOOL-X is used to conduct a preliminary analysis on the usage of failure models as valid alternatives to fault models, which may accelerate the injection process without losing representativeness.

Index Terms—dependability, fault injection, software faults, hardware faults, fault model, failure model, soft errors, tools

I. INTRODUCTION

The dependability of a system can be defined as “the ability to deliver service that can justifiably be trusted” [1]. The concept encompasses several dependability attributes and includes the notion of threats in the form of faults, errors and failures. Faults can take various forms, but the community’s focus revolves mostly around two types, which are hardware and software faults, due to their significant probability of affecting current software systems.

Hardware faults, particularly transient ones (i.e., soft errors), are on the rise due to the increase in the number of hardware components [2], [3], allied to the miniaturization of microprocessors [4]–[7] and the usage of energy saving techniques [7] (e.g., dynamic voltage and frequency scaling [8]). At the same time, evermore complex software systems are prone to residual software faults (i.e., bugs) that escape the software testing phase [9]. Software fault rates have been shown to be related with the total amount of source code lines, number of changed lines, and complexity of the system [10], [11], which are software features that have increased dramatically in recent years, with the consequent rise in the number of deployed bugs. Therefore, the evaluation of dependability and its attributes is of utmost importance, not only to critical systems but also to any software system.

To evaluate dependability, it is often necessary to perform *fault injection* (FI), which is a well-established technique that accelerates the process of fault activation in software systems

by deliberately emulating faults in a system. Through the usage of fault injection, the generation of failure data is greatly accelerated, producing in a few weeks the amount of data that would otherwise have taken years. Despite fault injection having been widely used for several decades, the reality is that researchers and practitioners often develop their own fault injection tools from scratch, since there is a limited number of these tools in the public domain that are capable of being easily applied to different types of systems, supporting multiple fault models or possessing a low learning curve.

This paper presents the TOOL-X¹ framework, which is intended to provide quick setup of fault injection experiments, while bringing support for fault injection in modern, state-of-the-art computer systems, such as those that use virtualization or belong to cloud computing deployments. TOOL-X caters to both novice and experienced users in the field of fault injection by including a simple and easy-to-use graphical user interface where fault injection campaigns can be setup, while at the same time allowing users to program complex campaigns. Users have at their disposal a range of pre-made components and templates that they can use in their campaigns and are able to develop and integrate their own components into the framework. Installation of TOOL-X has been made simple thanks to the use of containerization technologies, which enables its installation in just a few minutes using a single command.

TOOL-X has been made open-source and the link to the repository will be made available after acceptance. When compared with existing fault injection tools, TOOL-X is one of the few projects that natively supports fault injection in virtualized and cloud computing systems, features a graphical user interface and supports injection of both transient hardware faults and software faults out-of-the-box.

Thanks to its ability to integrate new components and tools, TOOL-X was used to investigate whether failure models (i.e., actual failure/wrong outputs of components instead of injection of faults) are a valid alternative to fault models when performing fault injection. In other words, we researched whether injecting failures (e.g., node crash, process crash, process hang) yields results similar to those obtained using representative fault models (e.g., single bit-flip in CPU registers) in less time. Our experiment focused on a popular cloud computing setup based around Openstack and used a workload composed by common operations that cloud administrators

¹Due to double-blind, TOOL-X is used as a placeholder for the real name

perform. A new fault injection tool was developed to inject failures, more precisely, process crashes, which was integrated into TOOL-X. The results suggest that failure models can be used to accelerate campaigns, however the resulting failures appear to differ from those obtained when performing fault injection using fault models, recommending some care with the representativeness of experiments that use failures modes.

The contributions of this paper include:

- 1) An open-source framework for easily conducting fault injection campaigns that supports different fault models including models representative of transient hardware and software faults;
- 2) A study on the viability of using failure models to accelerate the fault injection process.

The structure of the paper is as follows. In Section II we provide a detailed description of TOOL-X, including its architecture, its components, the fault injectors, its frontend and its installation process. Section III presents the preliminary experimental evaluation of failure models as an alternative to fault models, including a description of the setup, the results, observations and limitations. Section IV provides a review of the related work in the fields of fault injection tools and Section V concludes the paper.

II. FRAMEWORK DESCRIPTION

The TOOL-X framework was developed with ease-of-use and expandability in mind. It allows the novice user to quickly run new fault injection campaigns using the graphical interface, as well as the experienced user to design and tailor a campaign with the minimum amount of effort and knowledge on the details of the target system and on the fault injection process. To achieve this, the TOOL-X framework incorporates multiple out-of-the-box elements that implement specific functionalities and that can be mixed and matched according to the different needs.

TOOL-X is composed of two modules, Frontend and Backend, as shown in Figure 1. The Frontend module provides a graphical web interface where the users have access to the various functionalities of the framework. Through the graphical interface the users can register and login, view their campaigns, create new campaigns and perform a preliminary data analysis. The Frontend module was developed using React, as it contains libraries that implement various functionalities.

The Backend module consists of two software components: the Manager and a REST API. The Manager is the heart of the framework and is the software component responsible for executing the fault injection campaign and storing its results. The Backend module spawns multiple instances of the Manager process, one for each campaign being executed. The REST API exposes the functionalities of the Manager to the Frontend module. The Backend module also encompasses one database, where the information about the users and their campaigns is kept, as well as multiple CSV files that contain the results of each campaign. The Backend module is self-sufficient and can be used without the Frontend module. The

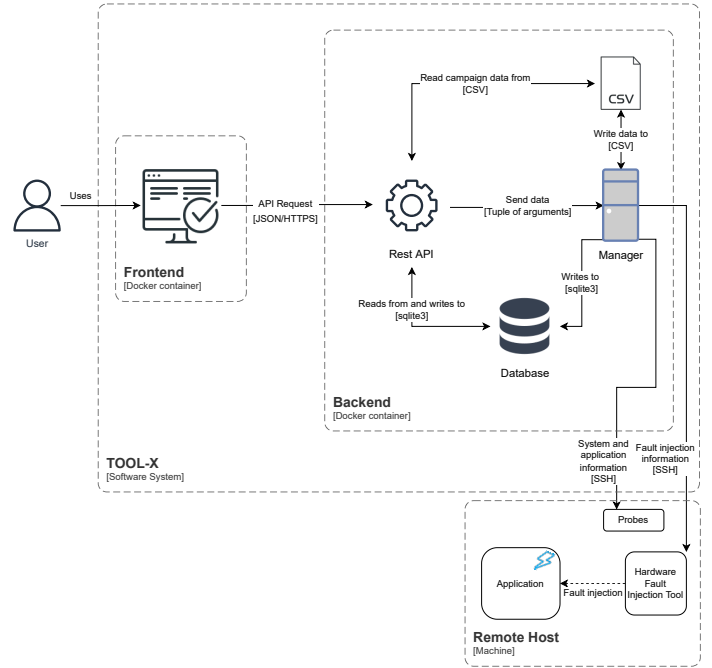


Fig. 1: Architecture of TOOL-X.

language adopted for the development of the Manager was Python 3.8, due to its simplicity and due to the range of libraries that are available. The technology used to develop the REST API was Flask and the database was SQLite, since the framework is aimed at smaller groups, for example research groups, hence a simpler and lighter engine was chosen.

A. Frontend & Graphical User Interface

The Frontend module provides the graphical interface of the framework, which includes web pages with various functionalities that are organized into four sub-modules:

- Authentication - Sub-module related to user authentication, e.g., login, registration and password change;
- Menu - Sub-module related to the display of information, such as campaign status;
- Campaign Menu - Sub-module related to the analysis of the campaign results, including campaign statistics and creation of graphs;
- Campaign setup - Sub-module related to the creation and configuration of new campaigns.

The first sub-module (Authentication) includes the functionalities of registering a new account, signing in and recovering/changing the password. Figure 2 shows 4 different pages: login, create account, recover password and change password. Although authentication is not a common feature in a fault injection framework, it enables multiple users to share the framework seamlessly.

After login the user sees the menu page, shown in Figure 3, where a list of the user's campaigns and associated information is presented. The user can search and filter campaigns by campaign name, execution name and campaign type name and change the number of campaigns to display at once.

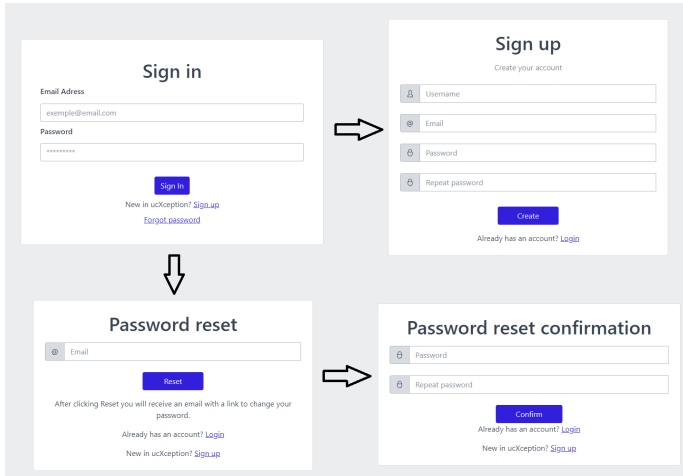


Fig. 2: Authentication page.

an informative message is used.

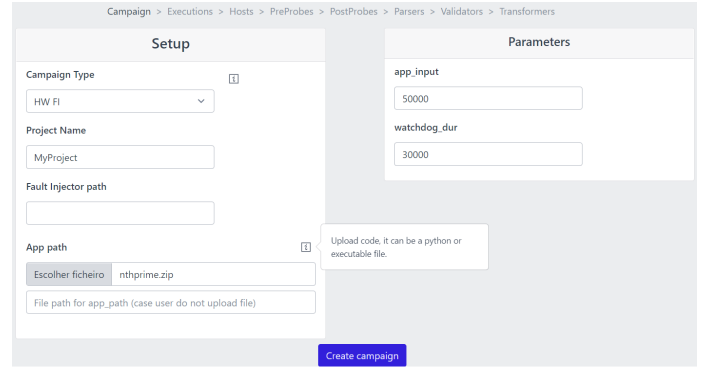


Fig. 5: Create campaign page.

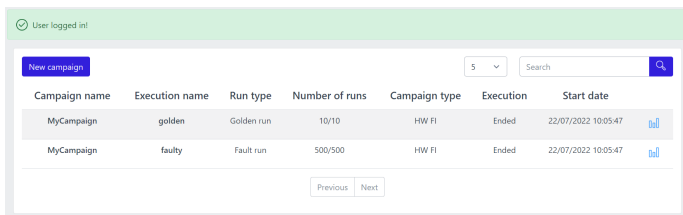


Fig. 3: Menu page.

After a campaign has been executed, the user can view a brief analysis of its results. The statistics page, shown in Figure 4, contains a statistical analysis of the campaign results, including information regarding run duration, failure percentage, incorrect content percentage, among others. Furthermore, a line graph in the page shows the evolution of the crash and incorrect data percentages as the number for runs increased, thus indicating whether the results have converged or not.



Fig. 4: Statistics page.

The campaign creation page has two different types of fields: configuration and parameters. The setup fields are associated to the general configuration of the campaign, like name, path to the injector, files to upload, while the parameters are related to the more specific configuration of each campaign. Depending on the selected campaign, the displayed parameters change. To aid the novice user to comprehend what is expected from each field, a help box which triggers a pop-up containing

B. Manager

The Manager, which is the core of the Backend module and which carries out the fault injection campaigns, is composed of a set of pre-made elements that provide contained functionality that can be connected together to solve the user's needs. According to the nomenclature adopted by TOOL-X, these elements can be classified as:

- Campaign - A campaign consists in a set of runs according to a specific configuration (which is stored in a database). Different runs can have different parameters, *e.g.*, some runs may perform injection while others will not (golden runs).
- Run - A run represents a single execution of the experiment flow defined in the campaign configuration, using the given run- and campaign-specific parameter values.
- Watchdog - A watchdog is used to monitor the execution time of a run and to ensure that it does not extend over the user-defined allotted time. If the run is taking too long and since it may even never end (*e.g.*, the application has entered an infinite loop), the watchdog will kill the workload application and record the occurrence.
- Probe - A probe represents an application that will be launched for the duration of the run and has the purpose of monitoring and storing information relative to the system or application being evaluated. Probes can be subdivided into pre- and post- probes, according to whether they are launched before or after the workload has started. Pre-probes usually collect system-wide metrics, whereas post-probes monitor specific processes, hence the need for post-probes to be launched after the workload.
- Fault injection tool - The fault injection tool implements a specific fault model (*e.g.*, software faults, single bit-flip for emulating transient HW faults) and allows the emulation of faults according to that model.
- Validators - The validators are small pieces of code, usually Python functions, that inspect the results obtained during a run and verify acceptance conditions. If a validator fails (*e.g.*, fault injection was not successful) then the data for that run will not be written to disk.

- *Parsers* - A parser reads the results of the run (*e.g.*, output of the fault injection tool, workload output) and converts them into a more useful and compact format. The various parsers write always to the results CSV file.
- *Transformers* - Transformers are similar to parsers, since both receive raw input and convert it into a processed output, but whereas parsers store their output in the results CSV file, the output from transformers is stored as individual files in the run's own result's folder. Transformers are mostly used to convert the raw output of the probes into a more manageable format, *e.g.*, converting a binary file originating from a resource monitoring probe into a CSV file where each row represents a time step and each column represents a monitored resource.

Since fault injection experiments can take place both in single-machine setups and in distributed systems, TOOL-X was designed to be able to transparently support both local and remote execution. The taken approach consists in defining a list of remote hosts, which includes the information required to perform a login via SSH (namely the IP and username). When required, every function in TOOL-X is capable of parsing the information regarding the host in which they should execute.

To facilitate the design and creation of new campaigns by an expert user, a base campaign template is provided. The flow of each experiment run is defined by the campaign configuration file, but usually obeys the flow that is provided by the base template and consists in the following steps:

- 1) *Launch pre-probes* - The pre-probes are launched and start to monitor their targets.
- 2) *Launch workload* - The workload is started. The user must programmatically define this step, possibly using the available utility functions.
- 3) *Launch post-probes* - The post-probes are launched. Normally the post-probes require the PID (or similar information) of the processes that they will monitor.
- 4) *Launch fault injection tool* - The fault injection tool (faultload) is launched. Per run only one injection is performed (unless explicitly modified in the code), in order to avoid a previous injection influencing future injections and thereby skewing the results. Although the fault injection tool is launched at this point, the fault may only be injected at a later moment, as the tool itself can have its own triggering mechanism. Unless modified, the values passed to the fault injection tool are randomly chosen from pre-defined valid ranges that will determine the actual type of fault injected in each run.
- 5) *Peak loop* - During this phase the workload executes, and the fault injection will take place at some point during its course. A watchdog process is launched with a configured pre-determined amount of time, if the workload does not finish within the allotted time, then the watchdog will forcefully kill the workload and the fault injection tool (if required). Otherwise this phase ends as soon as the workload process terminates.
- 6) *Post finish* - Usually consists, at least, in stopping the

probes, but may include other user-defined operations that should be executed right after the workload has ended.

- 7) *Extract data* - Extracts the data from the probes and stores them in the run's results folder.
- 8) *Launch transformers* - Launches the transformers that will read the stored data and convert it to another format, which will once again be stored in the run's results folder.
- 9) *Launch parsers* - Launches the parsers that will produce the output is stored in the main results CSV.
- 10) *Launch validators* - The last step consists in validating the results as to ensure their correctness.

At the end of each successful run (*i.e.*, when the validators do not flag a correctness error in the results), the data is added to a Pandas dataframe, which will be written to disk after the current campaign has ended. If the framework is stopped before the campaign has a chance to end, the data collected up until that moment is nevertheless stored to disk.

C. Components

TOOL-X provides a range of pre-made components that the user has at his own disposal. However, the expert user can create his own component and add it to the framework. Currently the following pre-probes are available:

- *Logs probe* - A simple probe that extracts logs from the target system during the *Post finish* phase. It is ready to extract logs specific to Linux, Xen and Openstack. The user can easily configure it to support other types of logs.
- *IntelPCM probe* - Intel PCM (Processor Counter Monitor) [12] provides a way of monitoring hardware counters in recent Intel hardware. This probe can be used to monitor the CPU, memory and power counters.
- *Ping probe* - A simple probe that performs pings at an user-specified interval between a source and a target computer. Can be used as a rudimentary way of monitoring the state of various systems.
- *SAR probe* - SAR [13] is an utility that uses the various interfaces provided by the Linux kernel to monitor system-wide activity information, such as CPU, memory, network, disk or power metrics. It takes a snapshot of all the available metrics at an 1 second interval (the lowest possible) and stores the results in a binary file.
- *TCPDump probe* - Monitors and stores all the network traffic in a specific interface. Supports passing TCP-Dump [14] rules to filter the packets that are captured.
- *Xentrace probe* - Xentrace [15] is an utility that monitors the events that occur in a Xen virtualized system. The results are stored in (usually large) binary files.

With regards to post-probes, only the following is currently available:

- *Pidstat probe* - Somewhat similar to the *SAR probe*, since it also captures similar metrics, but focuses on a specific process (whereas SAR is system-wide). Can be used to monitor the workload application.

In terms of parsers, the following are available:

- *HW FI parser* - Reads the output produced by the TOOL-X's HW fault injection tool, which emulates transient hardware faults, and stores the register, the bit, the injection time, the PID of the process that was affected by the injection and the pre- and post- injection values of every register.
- *SW FI parser* - Stores information relative to the injection performed by the TOOL-X's SW fault injection tool, such as the applied operator or in which line the fault was injected.
- *Pcap -> TCP parser* - Reads the data from a *TCPDump probe* and calculates statistics, such as, total packets, total packets by type (RST, FIN, ...), retries, and others.
- *Info parser* - Stores generic information about the run, such as its start time, end time and duration.
- *MD5 output parser* - Obtains the output of the workload application and computes its MD5 hash. Compares the obtained MD5 hash against a fixed, expected hash and records whether both hashes match and the size of the produced application's output. Useful to detect silent data corruptions whenever the workload application produces a deterministic output (*i.e.*, always produces the same output when it receives the same inputs).
- *Return code parser* - Stores the return code of the workload process. Can signal a successful termination or an abrupt termination (*e.g.*, killed by the operating system due to a segmentation fault).
- *Current folder parser* - Minimalistic parser that just stores the path of the results folder of the current run.

Concerning transformers, the following are available:

- *Pcap -> TCP 2 CSV transformer* - Converts a PCAP dump of network traffic into a CSV file with high-level information about each packet, such as the TCP flags, packet size, IPs and ports, or timestamps.
- *Pidstat 2 CSV transformer* - Converts the binary file generated by the *Pidstat probe* into a CSV file.
- *SAR 2 CSV transformer* - Employs the *sadf* utility [13] to convert the binary file produced by *SAR probe* into a CSV file.
- *Ping 2 CSV transformer* - Converts the output of the *Ping probe* into a more structured CSV file.
- *Save output transformer* - Saves the raw output (stdout and stderr) from the workload application into files. Can be used when a more detailed analysis to this output is required, or for debugging.

There is one available validator, called *Ensure Injection*, which checks whether one and only one injection (of the TOOL-X HW fault injection tool) has occurred in a run by comparing the pre- and post-injection values of all registers and ensuring only one bit of one register has changed.

D. Fault Injectors

TOOL-X comes equipped with three fault injection tools that implement different fault models. There are two different fault injection tools for emulating hardware faults, which focus

on different types of systems, and a tool for emulating software faults. Moreover, other fault injection tools can be integrated into the framework.

1) *Hardware faults in Linux-based systems*: This tool emulates soft errors that affect the CPU's register file or other components of the CPU (buses, ALU, FPU, *etc.*), by implementing the single bit-flip fault model [16], [17]. Bit-flips are restricted solely to general purpose CPU registers and there is no support for directly performing bit-flips in the memory. The decision of not including injections in memory words was supported by the existence and popularity of very effective ECC for memory and by the fact that part of the soft errors affecting the memory can also accurately be represented by injections in register files.

The tool can run in any modern Linux kernel and supports the x86_64 and ARM architecture. It employs the *ptrace* functionality available in practically every Linux installation and which is also the engine behind the famous *gdb* debugging tool, to attach itself to a running process, briefly suspend its execution, obtain the data structures of the Linux kernel that hold the process' register values, perform the bit-flip according to the passed parameters, and resume execution. After the target process resumes execution, its register values will include the bit-flip. Since the tool is software-only and does not depend on any hardware extension or feature, we are referring to SWIFI (Software-Implement Fault Injection). Furthermore, since the injection can be performed without requiring any modification to the target program's source or binary code, it can be classified as a run-time approach [18].

The tool also includes logging functionality that stores the exact timestamp of the injection moment and the values of every register prior and post the bit-flip. This information is extremely useful not only to validate that injection is working correctly, but also to enable detailed and complex analyses of the results.

The moment of injection is always temporarily triggered, but there is support for two ways of setting this trigger: *timeout* and *deadline*. In *timeout* mode the user specifies how many milliseconds the tool should after it is launched and before it performs the bit-flip. Whereas in *deadline* mode, the user specifies a UNIX timestamp (including milliseconds) which defines the desired moment of injection and which the tool will attempt to obey as closely as possible. Localization-based triggering, *i.e.* triggering the fault whenever a certain instruction is executed, is not currently supported, as this tool's approach is not the best candidate to support such a triggering mechanism. The flow of the this fault injection tool is hence as shown in Figure 6.

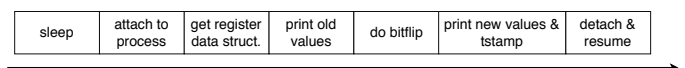


Fig. 6: Flow of TOOL-X's HW fault injection tool for Linux-based systems.

2) *Hardware faults in virtualized systems*: A separate fault injection tool capable of emulating hardware faults was created

specifically for use in virtualized systems. It is capable of injecting faults in any application running inside a VM, including a hypervisor as long as nested virtualization is used (*i.e.*, the hypervisor being targeted is executed inside a VM).

The fault model remains the traditional single bit-flip in CPU registers and any of the *rip*, *rsp*, *rbp*, *rax*, *rbx*, *rcx*, *rdx* and *r8* to *r15* x86-64 registers can be targeted.

The tool was implemented as a set of modifications to the Xen hypervisor, which introduce a new hypercall and respective toolstack functions to control the fault injection process, as well as modifications to the scheduling subsystem to enable injections of faults inside VMs.

The injection process consists in modifying the register value stored in the data structure that holds a VM’s CPU state and which is updated immediately prior to a context switch. This structure is needed because every hypervisor must know the latest state of the CPU between context switches of VMs. We take advantage of this fact to inject faults, but this means that the approach is dependent on the rate at which context switches occur, which is a configurable parameter in Xen. While higher context switching rates (*i.e.*, smaller timeslices) allow the fault injection tool to have a more precise moment of injection, they can also bring considerable performance overhead and intrusiveness to the system.

Furthermore, this tool is capable of filtering the application that is targeted for injection by looking at the value in the *rip* register (which points to the next instruction to be executed) and only performing injection whenever the *rip* is inside a user-defined range. This functionality can be specially useful if one wishes to perform fault injection that affects solely the hypervisor (or solely the non-hypervisor code) running in a VM, as there is a well established division between the virtual memory addresses assigned to the hypervisor, to the operating system and to the userspace applications.

Figure 7 presents the expected usage scenario for this tool.

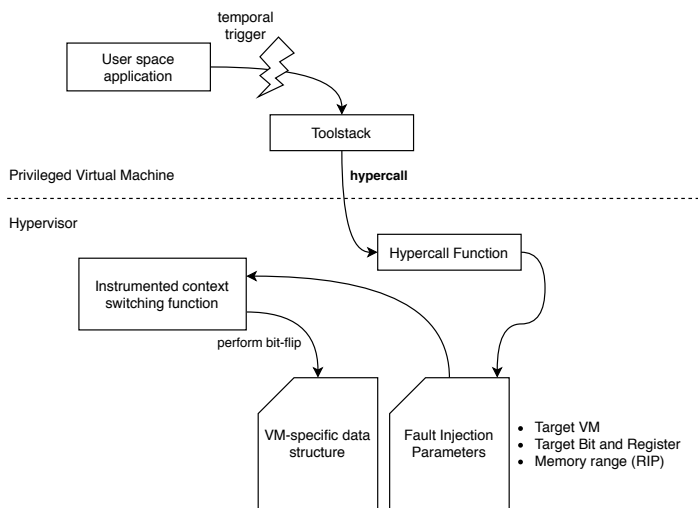


Fig. 7: Flow of TOOL-X’s fault injection tool for virtualized systems.

The flow starts from the privileged virtual machine (PVM),

also known as dom0 in Xen’s nomenclature, where the TOOL-X framework provides the triggering functionality, which is not embedded in the fault injection tool, and calls the toolstack at the correct moment. The toolstack will perform a hypercall to a function in the hypervisor, while passing the desired parameters for fault injection. These parameters include the target VM (when a system has multiple VMs, the tool can focus on just one of them), the target register and bit where injection will take place, and the start and end of the memory range that the *rip* should be pointing at if injection is to take place, although this last parameter is optional. The hypercall function will write this information to an internal structure, which will be read during context switching, and if all conditions are met (the VM that is receiving CPU time is the same as the target VM and its *rip* is inside the expected range) the bit-flip is performed here, right before the target VM starts executing.

3) *Software fault injection in C source-code*: Software faults are an important threat to the dependability of computer systems, including large scale and networked applications. Moreover, it is widely accepted that all computer programs contain software defects and, consequently, it is relevant to intentionally introduce defects as a means to evaluate how well a system is able to detect, isolate and recover from the ensuing errors.

TOOL-X supports software fault injection at the source-code level by applying program modifications that are representative of mistakes made by software developers [19]. Table I lists the software fault types that the tool is able to inject. The tool accepts programs written in the C programming language, widely used to develop system software.

TABLE I: Software fault types.

Operators	Description
MIFS	Missing <code>if</code> construct plus statements
MIA	Missing <code>if</code> construct around statements
MIEB	Missing <code>if</code> construct plus statements plus <code>else</code> before statements
MFC	Missing function call
MLAC	Missing <code>and</code> sub-expression in branch condition
MLOC	Missing <code>or</code> sub-expression in branch condition
MVAE	Missing variable assignment with an expression
MVAV	Missing variable assignment with a value
MVIV	Missing variable initialization with a value
WVAV	Wrong value assigned to variable
MLPA	Missing localized part of algorithm
WAEP	Wrong arithmetic expression passed in function call
WPFV	Wrong variable passed as parameter of function call

The faults listed in Table I are injected in programs written in the C programming language and the injector itself is written in Java, using the Eclipse CDT plugin, which is the plugin that supports C/C++ programming in Eclipse. The fault injector takes as input the source code and CDT performs lexical and syntactic analysis to produce the corresponding abstract syntax tree. The fault injector then searches the tree to identify the nodes in which faults can be injected. For each possible location/fault type pair, the tree is modified accordingly and the resulting program is then converted back

into source code representation. Then, a call to the `diff` tool generates a `patch` file for each fault that can be injected.

E. Containerization

One of the unique advantages of TOOL-X is its easy and quick installation process, which is accomplished through the use of Docker containerisation technology. Docker is an open platform for building, running, and managing containers on servers and the cloud.

Docker revolves around two basic concepts: images and containers. An image is a read-only template with instructions for creating a container, and a container is a runnable instance of an image. To publicly share images, Docker provides a repository, called Docker Hub. As such, TOOL-X consists of two images (one for the Frontend module and another for the Backend module) and we have made them publicly available in Docker Hub.

Regarding the environment configuration, the framework administrator can extract the image from the Docker HUB repository. Then the administrator must execute a command to generate a container of the respective image. The command will be practically identical for the creation of the containers for the two modules. Mainly, the administrator must define an IP address, a port and must also specify two crucial environment variables for the modules to work properly:

After pulling the images from Docker Hub, some parameters unique to each host system must be configured. These are the IP address and ports where TOOL-X will listen as well as two crucial environment variables:

- `REACT_APP_API_URL` - Holds the URL of the REST API. Required so that the Frontend module can send requests to the API.
- `FRONT_END_URL` - Contains the URL of the frontend web page. The API needs to know the web page address, because when an email is sent due to a password change request, a link is also sent which redirects the user to a specific page of the Frontend module.

As stated, the framework can execute operations, inject faults and collect data from remote systems. However, in order for this feature to be operational, a private/public keypair should be shared across the containers and every remote host that will be used for the campaigns. For the images published in Docker Hub, a pre-defined keypair has been used.

If the expert users wish to define their own keypairs, they can do so, but such implies manually creating the Docker images. Fortunately, creating new images in Docker is simple and depend on editing a Dockerfile, which is a file is used to define the steps required to create an image. This Dockerfile can be edited so that TOOL-X uses a different keypair, as well as to add new components or campaigns.

III. EVALUATING FAILURE MODELS FOR CAMPAIGN ACCELERATION

Fault injection using fault models has been widely used for evaluating the dependability of systems and to validate fault tolerance mechanisms. However, despite being effective, it is a

slow process because many faults do not have any effect (i.e., do not cause any visible failure in the target system). Fault injection using failure models can, hypothetically, be able of reproducing the same results, with similar levels of accuracy and representativeness, but at a fraction of the time and cost.

Although failure models have been used before, to the best of our knowledge, no study has verified whether the produced results are representative nor that failure models bring a speed and cost improvement. In this study, we will perform fault injection using both fault and failure models and compare the obtained results in order verify and validate the aforementioned points.

For the experiments we will use the TOOL-X framework and take advantage of its extensibility to integrate a new fault injection tool that uses a failure model and a new parser. We chose an experimental setup representing a cloud deployment and opted to use Openstack, a cloud operating system, as the target system for the experiments. Openstack is divided into several services to allow users to use the components according to their need, such as, compute, storage, networking, orchestration, shared services, among others.

A. Setup

In order to run the experiments a physical setup was configured. Its specifications in terms of hardware and software are given in Table II. The TOOL-X was installed on the machine manually, thus without using any containerization technology.

TABLE II: Experimental setup specification

Component	Description
Operating system	Linux 4.14.89
Hypervisor	Xen 4.11.1
CPU	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
CPU(s)	4
Thread(s) per core	2
RAM	16 GB
Disk1	1T*7200 RPM
Disk2	10T*7200 RPM

Openstack contains a plethora of services that are optional. In our setup, the 3 most common Openstack services were configured. One service (Nova) supports the creation of virtual machines and provides an API and tools for managing the resources of the cloud. Another service (Neutron) provides "network as a service" between interface devices managed by other Openstack services, such as Nova. The hypervisor used in Neutron to host the virtual machines was KVM/Qemu 4.2.1. Finally, the third installed service (Cinder) is a block storage service and is designed to present storage resources to end users that can be consumed by Nova.

Two pre-probes were configured to collect metrics regarding the three configured services. For Nova, the *Logs probe* was used to extract logs from the target system, in this case Openstack, and the *Ping probe* was configured to perform pings on the three services to monitor the various systems.

Figure 8 illustrates how the setup is configured and how fault collection and injection is performed. TOOL-X executes the workload and launch the probes for each service installed.

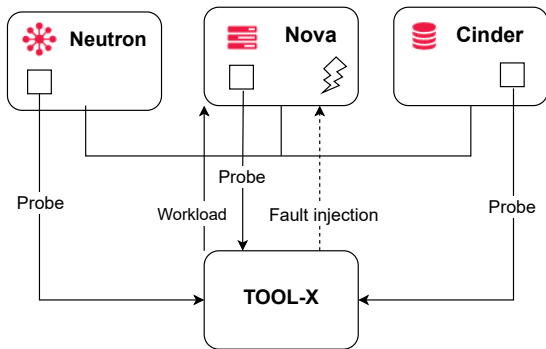


Fig. 8: Experimental Setup.

While the workload is being executed, the TOOL-X manages the fault injection process.

B. Workload

The workload consists of several types of requests made to Openstack, which represent some of the most common operations that a system administrator might perform, such as listing, creating and deleting flavors or instances. The workload follows a sequential flow which performs, in total, 11 operations, although some are repeated more than once. The operations are:

- 1) List all flavors (i.e., the resource configurations that can be used by the virtual machines)
- 2) List all instances (i.e., virtual machines)
- 3) Create a new flavor with a certain configuration
- 4) Create a new instance using the previously defined flavor
- 5) Delete a flavor
- 6) Delete an instance

The workload is simplistic and can be extended in the future to perform other operations. In total, the workload takes on average 130 seconds to finish.

C. Injection process

Regarding the injection process, we defined what, where and when to inject based on our setup and the goals of our study. The objective was to use fault and failure models to emulate transient hardware faults affecting a process of Openstack. Software and other kinds of faults were not considered.

Regarding the fault model, the single bit-flip fault model was used. This model emulates soft errors affecting the CPU register file directly or other CPU components (buses, ALU, FPU, etc.) indirectly. Both the bit and register are chose randomly following an uniform distribution, once per run.

For the failure model, we injected crashes of a process, which is a common failure mode obtained in fault injection experiments where a single bit-flip is injected in a random CPU register. Other failure models can be evaluated, however we chose to begin with this one because it is simple to implement and emulates a large portion of failures.

A new tool was developed which randomly chooses and kills (by sending a SIGKILL) a random process of nova-api, which is the service that receives the operation requests and

passes them on to the correct service to be handled. Due to the extensibility of TOOL-X it was possible to add the new failure model injection tool that was developed later to the framework.

One of the various Openstack-related services of the Nova VM was chosen to be the target, as Nova is the central element of the setup. In the future we plan to conduct similar experiments in the other services of Nova and of the other Openstack components. As the workload takes about 130 seconds to execute, the injection time was set to the range between]10,100[seconds, chosen randomly. The first 10 seconds correspond to the warmup and the last 30 seconds are the cooldown, which allow the injected faults to manifest and cause failures.

D. Failure detection and classification

The expected output from running the workload includes information about the return code of each operation (which indicates whether the operation executed successfully or not), the duration of its execution and, for some operations, the output produced by the operation. These parameters allow a detailed analysis to be made of the output during the experiments, with the aim of assessing whether the result after fault injection remains as expected. For automatically performing the data processing, a parser was developed and integrated into TOOL-X. For each operation of the workload, the parser processes the workload output and writes the following data to the CSV file: Correct or incorrect output and the respective output size, status code, total time taken to perform the operation.

Another measure that must be considered, but that is not treated by the parser, is the watchdog parameter that is set at the beginning of the campaign configuration. As the execution of the workload takes about 130 seconds, a watchdog of 200 seconds was defined. This time is longer than the execution of the workload so that the workload has time to finish by itself. If the workload takes longer, then the watchdog terminates the process as to avoid waiting for a possibly hanged process.

These metrics allow the results to be classified in two ways: 1) No Effect (the workload executes seemingly without problem), and 2) Failed (at least one operation returned a status code different from expected, thus signaling an unsuccessful operation). Silent data corruption, despite important, was not included because no occurrence was detected.

E. Results

A total of 1000 runs were executed, equally distributed between injection using fault and failure models. Applying the classification scale resulted in 98.2% of No Effect for the failure model and 92.8% for the fault model. Therefore there were 10.6% of failed runs when using the failure model and 3.4% with respect to the fault model.

Figure 9 shows a comparison between the amount of operations that failed in a single run for fault and failure models. It should be noted that the Y-axis is zoomed in between 0 and 6%. An analysis of the results concludes that not only

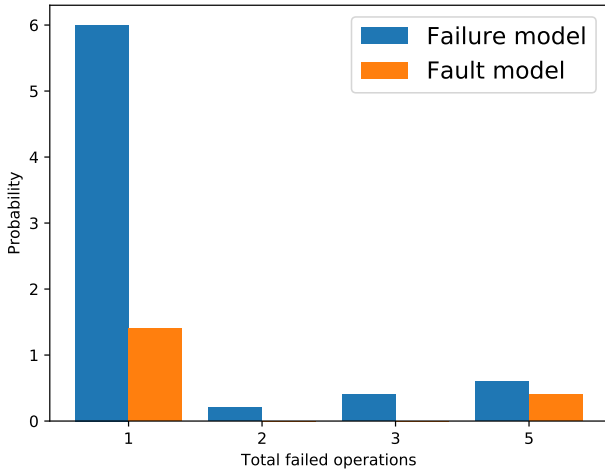


Fig. 9: Failed operations per run for both models.

failure models cause more failures, the generated failures also affected more operations, on average.

Figure 10 shows the probability of an error occurring in each of the various operations of the workload. Once again the Y-axis is zoomed in. The graph shows that operation T7 has a higher failure probability, when using both fault and failure models, which can be explained by being a more sensitive operation or by the moment of injection. The two first and last operations did not experience any failure likely due to the warmup and cooldown periods.

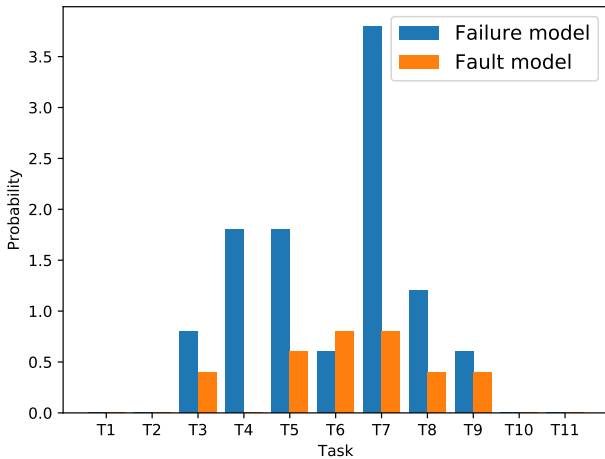


Fig. 10: Failures per operation for both models.

To measure how the failure probability evolves when using failure models, Figure 11 plots the distance between the average failure probability obtained in each run when using failure models against the final failure probability obtained using fault models (i.e., 3.4%). It can be seen that the distance begins to stabilize after around the 200th run, but tends to slightly increase as the runs increase.

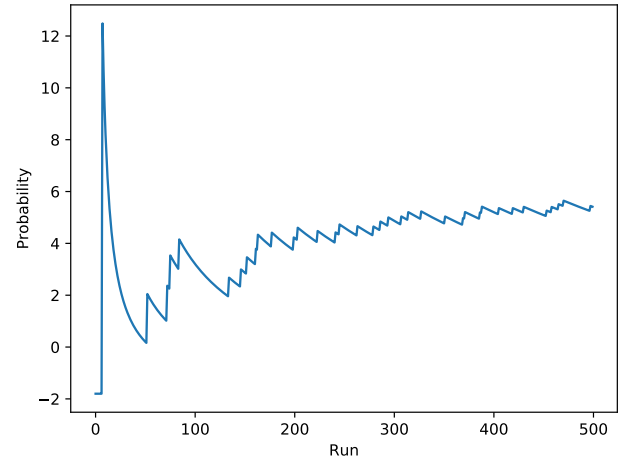


Fig. 11: Distance between failure model and oracle.

F. Observations & Limitations

Despite the exploratory nature of this experimental evaluation, some observations can already be made. Nevertheless more experiments are needed to reach acceptable confidence. One observation is that the usage of failure models produces more failures than the usage of fault models, thus accelerating this type of experimental campaigns. This is to be expected after all, but carries implications to practice. Namely, if the objective of a campaign is to quantify the failure modes and their probabilities of a system, the usage of failure models should be complemented with a correction factor, otherwise the obtained probabilities will be higher than what they really are. On the other hand, if the objective of a campaign is solely to collect failure data or to validate a fault tolerance mechanism, then failure models appear to be a valid and more efficient choice.

Another observation is that the failures obtained when using failure models appear to differ from the failures generated by injecting faults. This can be seen in Figure 10, where the distribution of failures across different operations appears to not follow the same pattern. For example, failure models caused failures that strongly affected operation T7 whereas fault models did not. Another example is in operation T4, which experienced failures when using failure models but never when using fault models.

Although some progress has been made in the comparison between fault and failure models, the present study suffers from various limitations that will be addressed in future work. First of all this experiment focuses only on a specific experimental setup, thus the observations cannot be extrapolated to other setups. Another limitation is related to the workload, which exercises only a small, yet often used set of operations and which does not produce an elevated load on the system's resources. Other workloads should be considered as part of future experiments. Different faultloads, such as different fault models, can also be considered. Finally, the obtained number of experiment runs is still relatively low.

Even though limited, the results support carrying out further

experiments in this topic and show how the extensibility of TOOL-X can be an advantage when researching topics that fall out from the more traditional use cases associated with fault injection.

IV. RELATED WORK

Given that fault injection is a mature technique with decades of academic and commercial use, several fault injection tools and frameworks have been described in the literature. When compared to these, TOOL-X constitutes a more recent framework that has support for multiple fault models and that can inject in virtualized and cloud computing systems.

In this section, a brief description of other important fault injection tools is provided following a chronological order. MESSALINE [20] is a ‘general physical-fault injection tool’ composed by four modules: fault injection, target activation, readout collection, and management. It supports a range of fault models, which include stuck-at-0 and stuck-at-1.

FIAT (Fault Injection-based Automated Testing) [21] has a structure composed by fault injection manager, which controls the experiments, and a fault injection receptor, which receives the results for posterior analysis, and includes the ability to support distributed systems, monitoring the systems and injecting faults through a software-implemented compile-time approach, according to a fault model of single or multiple memory bit-flips.

FERRARI [22] is a fault injection tool that injects faults by modifying the process’ injection state through the *ptrace* functionality, very similarly to the approach taken by one of the fault injection tools of TOOL-X that has been described in this paper. FERRARI is capable of injecting memory corruption faults, or transient and permanent faults, supports time and location triggers, and has five different fault models, which are bit-flips, bit setting, bit resetting and byte setting.

FINE [23] (Fault Injection and moNitoring Environment) supports injection of software faults and hardware-induced software errors into UNIX kernels. It is composed by four parts: the fault injector, the workload generator – which generates a workload of systems calls, the controller, and the software monitor – which tracks variables of the kernel and its control flow and stores this information to disk.

Xception [24] employs a hybrid approach which combines software-implemented fault injection with debugging and performance-monitoring hardware extensions as to reduce the overhead of the fault injection process and to monitor fault activation and the target system. It supports fault models such as bit-flips, stuck-at-0 or stuck-at-1, on main memory and the processor, and is capable of performing triggering by time or upon specific instructions.

NFTAPE [25] is a framework for fault injection that can be used in various types of systems and supports multiple fault models, including bit-flips in registers and memory, communication errors and I/O faults, and multiple fault triggers, including spatial, time-based and event-based.

Goofi-2 [26] is capable of using both hardware-implemented and software-implemented techniques to emulate transient

hardware faults in CPU registers and memory using single and multiple bit-flips. It supports three different fault injection techniques: Nexus-based, exception-based and instrumentation-based. All these techniques are in one way or another dependent on features of the underlying hardware.

Gigan [27] is a software-implemented fault injection tool capable of introducing faults in memory and CPU registers of a virtualized system as single bit-flips that are triggered using breakpoints.

LLFI [28] is a fault injection tool that operates at the intermediate code level of LLVM in order to inject hardware faults in a compile-time manner which supports specifying the location of the fault. The propagation of the fault can be traced through the application using instrumentation in the program.

Marcello Cinque and Antonio Pecchia introduced a fault injection framework aimed at virtualized multi-core systems [29]. Their framework emulates hardware errors by modifying the values of special registers that belong to the Machine Check Architecture (MCA), in doing so they are able to evaluate the error handling mechanisms of the system.

CloudVal [30] is a framework based on NFTAPE that was developed to validate the reliability of virtualized environments. It supports emulation of soft errors and injection of faults mimicking delayed I/O operations and maintenance events. Its fault injector was implemented as a loadable kernel module and features a spatial-triggering mechanism based on breakpoints.

V. CONCLUSION

In this paper, we presented TOOL-X, an open-source framework for orchestrating and conducting fault injection campaigns. TOOL-X is easy to install and to use and can be extended with new components and tools. It comes equipped with a range of components for monitoring the system-under-test and can emulate both software and transient hardware faults. TOOL-X was designed to be used in both local and distributed systems, particularly virtualized and cloud computing systems. As such, TOOL-X is one of the few projects supporting fault injection of different fault models and focusing in cloud computing and virtualized systems that has been made publicly available.

Using TOOL-X, an evaluation on the viability of using failure models as alternatives to fault models when performing fault injection was carried out. The results confirm that failure models can produce failures more frequently than fault injection, however the resulting failures may differ from those that occur when fault models are used. Further research is needed before a strong conclusion can be taken regarding this matter, which is planned as future work.

REFERENCES

- [1] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan 2004.
- [2] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner, “United states data center energy usage report,” 2016.

- [3] J. Koomey, "Growth in data center electricity use 2005 to 2010," *A report by Analytical Press, completed at the request of The New York Times*, vol. 9, 2011.
- [4] P. Hazucha and C. Svensson, "Impact of cmos technology scaling on the atmospheric neutron soft error rate," *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2586–2594, Dec 2000.
- [5] S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, vol. 19, no. 4, pp. 23–29, July 1999.
- [6] R. Baumann, "The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction," in *Digest. International Electron Devices Meeting.*, Dec 2002, pp. 329–332.
- [7] V. Chandra and R. Aitken, "Impact of technology and voltage scaling on the soft error susceptibility in nanoscale cmos," in *2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, Oct 2008, pp. 114–122.
- [8] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott, "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling," in *Proceedings Eighth International Symposium on High Performance Computer Architecture*, Feb 2002, pp. 29–40.
- [9] 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.
- [10] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88.
- [11] 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.
- [12] I. Corporation, "Intel PCM," <https://github.com/opcm/pcm>, 2019, accessed: 2019-02-01.
- [13] S. Godard, "SYSSTAT," <http://sebastien.godard.pagesperso-orange.fr/>, 2019, accessed: 2019-02-01.
- [14] Tcpdump, "TCPDump," <https://www.tcpdump.org/>, 2019, accessed: 2019-02-01.
- [15] D. Faggioli, "Tracing with xentrace and xenalyze," <https://blog.xenproject.org/2012/09/27/tracing-with-xentrace-and-xenalyze>, 2012.
- [16] R. Johansson, *On Single Event Phenomena in Microprocessors*, 1993.
- [17] G. L. Ries, G. S. Choi, and R. K. Iyer, "Device-level transient fault modeling," in *Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing*, June 1994, pp. 86–94.
- [18] R. Barbosa, J. Karlsson, H. Madeira, and M. Vieira, *Fault Injection*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 263–281.
- [19] 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, Nov 2006.
- [20] J. Arlat, Y. Crouzet, and J. . Laprie, "Fault injection for dependability validation of fault-tolerant computing systems," in *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, June 1989, pp. 348–355.
- [21] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Ysskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "Fiat - fault injection based automated testing environment," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995. ' Highlights from Twenty-Five Years '.*, June 1995, pp. 394–.
- [22] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: a flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 248–260, Feb 1995.
- [23] W. . Kao, R. K. Iyer, and D. Tang, "Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1105–1118, Nov 1993.
- [24] D. Costa, H. Madeira, J. Carreira, and J. G. Silva, *Xception™: A Software Implemented Fault Injection Tool*. Boston, MA: Springer US, 2003, pp. 125–139.
- [25] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer, "Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*, 2000, pp. 91–100.
- [26] D. Skarin, R. Barbosa, and J. Karlsson, "Goofi-2: A tool for experimental dependability assessment," in *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, June 2010, pp. 557–562.
- [27] M. Le, A. Gallagher, and Y. Tamir, "Challenges and opportunities with fault injection in virtualized systems," in *1st Int. Workshop on Virtualization Performance: Analysis, Characterization, and Tools*. Citeseer, 2008.
- [28] A. Thomas and K. Pattabiraman, "Llfi: An intermediate code level fault injector for soft computing applications," in *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.
- [29] M. Cinque and A. Pecchia, "On the injection of hardware faults in virtualized multicore systems," *Journal of Parallel and Distributed Computing*, vol. 106, pp. 50 – 61, 2017. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0743731517300849>
- [30] C. Pham, D. Chen, Z. Kalbarczyk, and R. K. Iyer, "Cloudval: A framework for validation of virtualization environment in cloud infrastructure," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, 2011, pp. 189–196.