MSc in Informatics Engineering
Dissertation
Intermediate Report

Evaluate the robustness of the Cloud

Gonçalo Silva Pereira
gsp@student.dei.uc.pt

Supervisor:
Raul Barbosa

Co-Supervisor:
Henrique Madeira

July 5, 2015

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Dedication

Acknowledgements

I would like to thank Thomas Corbat and professors Raul Barbosa and Henrique Madeira, who are role models, for their support and help in order to make good decisions.

I also would like to thank my girlfriend for her support, understanding and the fellowship along this path. To my friends and colleagues in the Department of Informatics Engineering for the patience and for all the times they have given me support.

Last but certainly not least, I would like to thank my family for the encouragement, love and all the unconditional and constant support that lets me fulfill this dream. Obrigado!

Gonçalo Silva Pereira

“ Bridges are normally built on-time, on-budget, and do not fall down. On the other hand, software never comes in on-time or on-budget. In addition, it always breaks down.

Alfred Z. Spector, Google Research ”

“ I have no special talents. I am only passionately curious.

Albert Einstein ”

Contents

List of Figures

List of Tables

Abbreviations

API Application Programming Interface

BPaaS Business-Process-as-a-Service

DDOS Distributed Denial of Service

EMP Electromagnetic pulse

HWIFI Hardware Implemented Fault Injection

IaaS Infrastructure-as-a-Service

ODC Orthogonal Defect Classification

PaaS Platform-as-a-Service

SaaS Software-as-a-Service

SWIFI Software Implemented Fault Injection

Abstract

The Cloud Computing is a new paradigm that provides on demand self-service resources, broad network access, resource pooling, rapid elasticity and a measured service through four different models, community, hybrid, private and public.

The main objective of this paradigm is to allow users to get the most of the technology without having the knowledge and skills to ensure the proper functioning of all the technologies involved, allowing the users to focus on their core business, rather than be blocked due to the technological difficulties.

The fact of the virtualization be the fundamental technology that powers cloud computing, provide the reduction of IT cost while increase the efficiency, utilization and flexibility of their existing computer hardware.

However, the Cloud Computing is not free of external disturbance like security attacks, power surges, workload faults, hardware and software faults. Due to this reason, the theme of my dissertation is "Evaluate the robustness of the Cloud" and it is based on the development of a fault injector software, to, as the name suggests, inject faults in software to testing it in the cloud later. After the testing, the collected results will be evaluated using the CRASH Scale.

Keywords: Robustness, Cloud Computing, Faults, Errors, Failures, Vulnerabilities, Fault Injection, Fault Tolerance.

1 Introduction

This internship deals with the challenge of assessing the robustness of cloud platforms. The computing service provider uses virtualization to manage and allocate computing power to meet present needs of the application. Faults will be injected in software running in the cloud and collected results will be evaluated.

1.1 Contextualization

The present dissertation describes the work developed in the scope of Master's degree in Informatics Engineering. It is focused on evaluating the robustness of the cloud computing (usually simply called "the cloud" or "cloud"), which is a very important issue nowadays, mostly because of its increasing usage. It is characterized by the placement of data and software and services on remote infrastructures. Despite its numerous benefits, the reliability of these platforms has not kept the needs, and users trust on their applications to systems outside of personal control.

In this context, the problem of the existence of bugs in software of the entity managing the platform where applications are executed arises naturally. There are many reasons for the existence of these software bugs, the most important being:

- Miscommunication;
- Software complexity;
- Programming errors;
- Changing requirements;
- Time pressures;
- Egotistical or overconfident people;
- Poorly documented code;
- Software development tools;
- Obsolete automation scripts;
- Lack of skilled testers.

Therefore, the bugs will continue to exist and will always exist. Because of this, the test of the ability of a critical system to deal with existing bugs is critical. This is the main reason for the existence of this dissertation.

Any organization that puts an application in the cloud (for instance in Microsoft Azure or Amazon EC2) should accept the assurances given by the service provider.

Although there are solid virtualization platforms, fault tolerance is still a problem in research. The system's ability to recover from the failures existence, named resilience, is a critical factor in the cloud.

1.2 Objectives

The main objective of this work is to evaluate the robustness of the cloud. To do that, I will design and implement a tool to inject software faults in the source code of some applications.

Nevertheless, this main objective is divided in some other goals:

- Implement the thirteen operators specified by João Durães;
- Use the fault injector to emulate faults in applications;
- Measure the time and the value obtained after running the application, in normal conditions;
- Inject a fault in an application, verify and analyze the effect. Measure the value and the time running the application with and without faults;
- Compare the time and value in a normal scenario and in a scenario with faults;
- Create a scenario with multiple virtual machines, verify and analyze the effect. Measure the value and the time with the application without faults and with faults;
- Compare all the results and obtain conclusions.

1.3 Document structure

In this document are specified all the related subjects with the project.

The second section presents the state-of-the-art in the related areas with particular emphasis to the fault injectors of software faults.

The third section is an important section of this report, because of the research involved in the execution of this work. It was necessary to take some important decisions based in research results, knowledge and my own experience.

The fourth section describes the work that has been done in Fault Injector, and the work that should be done in the next semester.

The fifth section explains other modules that need to be executed in this project to observe and evaluate the results of the fault injector.

In the last section, I will do an overview analysis of my work, in general the operators and the constraints developed. I will also talk about the work to be done in the next semester.

1.4 Management

1.4.1 Meetings

About the meetings, the supervisor Raul Barbosa and I agreed that meeting once every week was the best option. Moreover, they happened, with one or another change of schedule to reconcile with the other activities from both. In addition, I attended some general meetings of the project. In them, we could discuss concepts and the direction of the project with colleagues and teachers, among them: Raul Barbosa (supervisor), Henrique Madeira (co-supervisor), João Durães and João Fernandes.

1.4.2 Risks

As any other project, this one has risks too. Some of the risks are related to equipment failure and data lost. To prevent these situations I use *GitHub* to backup all the sources developed that are related to the project and this report. These backups are done whenever improvements are made to this project.

The particularity of the subject of this project is under intense investigation nowadays, bringing another kind of risks to this project, associated to the publication of similar research. To reduce this risk, I will check with regularity electronic publications, and if similar research was published, I will modify the project to assure that it adds additional value and it is not just like any other.

Moreover, I can have personal issues interfering with the progress of this project or lose the interest, and to prevent this I have selected a motivating topic at the beginning and I talk to the supervisor whenever I have doubts.

These risks, the preventative measures and the recovery measures can be seen, in other perspective, at Appendix B.

1.4.3 Planning and tracking

In Appendix A, is showed the Gantt diagram with the tasks that have been done during the first semester. As I postponed this dissertation for six months so, the scope and the context have changed. Now the two Gantt diagrams are incomparable.

About the development of this project, I have used an *Agile Life Cycle* based in an *Incremental Model*. New tasks are planned weekly, although it will always be a long term goal. The use of this type of methodology is important because it easily plans the following tasks to overcome any difficulties that have appeared.

2 State of the art

Nowadays, people use many services based in the cloud and many companies choose to use them too. By doing that, companies reduce the costs of IT infrastructure and not even need to buy "physical storage", neither care where the data is. The cloud service provides that the data is secure. However, such as any other computer system, the cloud has problems, like software and hardware faults, making the resilience of the cloud a very important issue.

The increased use of the cloud is related to a low usage of many dedicated servers, lower voltage levels, reduction of noise margins, increasing clock rates and because the cloud provider offer resources ready to deliver[1].

There are many studies showing that software faults[2] are the main cause of computer failures. Nevertheless, the number of faults that can be emulated is directly related to the technique used.

	Software	**Hardware**
Hardware		HWIFI
Software	SWIFI	SWIFI

Table 1: *Fault injection techniques and emulation environment.*

In the Table 1, it is possible to view that SWIFI techniques can be used to make software emulate software and hardware faults. Similarly, HWIFI techniques can be used to emulate hardware faults through hardware.

- **Software Implemented Fault Injection (SWIFI)** - the goal of this technique is to emulate errors at software level that happen during the execution environment, in hardware or software. Examples: Data corruption in registers, memory or hard drive; Communication problems in network or NoC; Software faults in binary code, in object files or in source code.

- **Hardware Implemented Fault Injection (HWIFI)** - this technique is related to the fault injections in the final system hardware. Examples: Electromagnetic pulse (EMP), radiation, etc.

SWIFI is an attractive technique because it won't require additional hardware (which would increase the cost of the test). The targets of this technique are the applications and the operating systems, but this technique has also some disadvantages: it cannot inject faults in inaccessible areas of software and it may disrupt or change the workload of the testing software. This technique can be used at:

- **Compilation time (object code level)** - Modify the structure of a program before the creation of an executable file;

- **Execution environment (binary code level)** - Changing the binary code activated by a timeout, an exception or a trap. At this level, less than seventy percent of the software faults can be emulated [3].
- **Before compile time (source code level)** - Change the source code by removing, replacing or inserting some simple code before the program compilation. At this level, all the software faults can be emulated;

I had the opportunity to access an application that inject faults before compile time (at source code level), named SAFE. I will describe it in next section, as well as others.

2.1 Software implemented fault injection of software faults

Below, I will describe some tools that use SWIFI techniques and that have made some improvements in this area of research.

2.1.1 JACA Tool

JACA[4] is a tool that has been made to validate Java applications. It injects high-level software faults and is based on computational reflection to inject interface faults in Java applications[5].

2.1.2 J-SWFIT

Java Software Fault Injection Tool[6] is a tool that does not need the source code to perform the injection, the mutation of the code is performed directly at byte-code level.

2.1.3 SAFE by Robert Natella

Safe is an application to inject realistic software faults in programs coded in C and C++. This tool uses MCPP as parser, to get the tree of code. The decision of using MCPP instead of GCC parser was a workaround for some of the shortcomings of the GCC's C preprocessor.

After that, some variations of original files are written (code with simple mutations) with the operators applied. Robert Natella implemented thirteen operators in SAFE, the same number as João Durães[7], but Robert implemented them at source code level, and João at binary level.

The fault injector under development will be similar in terms of output: source code files with changes made in it. However, its creation is justified since we do not have access to the code developed by Robert Natella, while this fault injector will be more maintainable and easy to use, using Java Language and not involving the MCCP prepocessor, which is already outdated.

2.2 ODC Model

Orthogonal Defect Classification [8] Model is a framework developed by IBM [9], created to improve the level of technology available to assist the decisions of a software engineer, by measurement and analysis. ODC can be used to classify and analyze defects during software development.

This model has eight categories:

- **Function** - This defect affects significant capability, end-user features, product Application Programming Interface, interface with hardware architecture, or global structure(s). It would require a formal design change.
- **Assignment** - Typically, an assignment defect indicates an initialization of control blocks or a data structure.
- **Interface** - Problems in the interaction with other components, modules, device drivers, call statements, control blocks, or parameter lists.
- **Checking** - Based on the program logic that is checked and failed to validate data and values before the usage, loop conditions, etc.
- **Timing/serialization** - Errors that happen in shared and real-time resources.
- **Build/package/merge** - Errors that occur in the integration of library systems, management of changes, or in version control.
- **Documentation** - Errors in the documentation can be propagated to publications and maintenance notes.
- **Algorithm** - Problems that can be fixed by re-implementing an algorithm or local data structure, include efficiency or correctness that affects the task.

João Durães used this model in his field-study, as a starting point for fault classification.

2.3 CRASH Scale

After the compilation and execution of the programs, the results need to be evaluated. To measure that, I will use the *Koopman's CRASH Scale*[10]:

- **C**atastrophic - Operating System crashed or multiple tasks affected;
- **R**estart - Task or process hangs, requiring restart;
- **A**bort - Task or process aborts abnormally (i.e. "code dump" or "segmentation violation");
- **S**ilent - Test Process exits without an error code returned when one should exist;
- **H**indering - Test Process exits with an error code not relevant to the situation or incorrect error code returned;
- **P**ass - The module exits properly, possibly with an appropriate error code.

The order of the letters in the word CRASH represents the impact to the operating system (Catastrophic is the worse and Hindering the least severe).

This *CRASH Scale* is a way to group the results of the effect of faults on an end-use system, mainly from the operating system perspective, by its severity.

3 Research objectives and approach method

In this section are discussed the main aspects in study.

3.1 Cloud Computing

To understand a little more what the Cloud Computing means:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."[11].

Cloud Computing is a new way to delivery IT services on-demand (utility-oriented and Internet-centric). These services include all the computational power: from hardware infrastructure as a set of virtual machines to software services as development platforms and distributed applications.

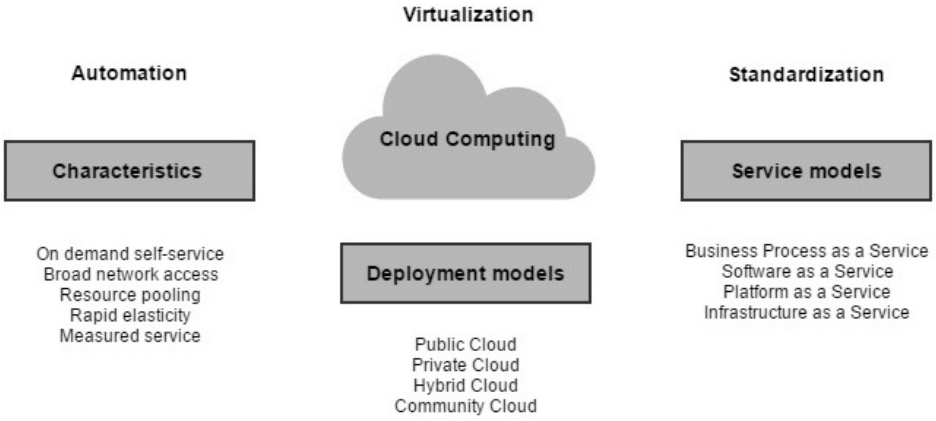

Figure 1: *Cloud computing overview.*

Below, Cloud Computing is described in relation to its characteristics, deployment models and service models[12].

Some characteristics of Cloud Computing are:

- **On demand self-service** - Users can request and manage their cloud computing resources without requiring human interaction, over a web-based self-service portal.
- **Broad network access** - Provide access over the network, through using standard way by several customers (e.g., mobile phones, tablets, laptops and workstations).

- **Resource pooling** - The computer resources are pooled to serve multiple customers through the safe separation of the resources at logical level.
- **Rapid elasticity** - Capability of resources to be elastically provisioned and released. Making sure that the application will have exactly the capacity that it needs at any point in time.
- **Measured service** - The service is monitored, measured, and reported transparently based on its usage. The costumers pay in accordance with the service spent.

Four models of deployment:

- **Private Cloud** - It is a single-tenant cloud solution utilizing client hardware and software, is located inside the client firewall or even in a data center. The sensitive information is maintained inside the organization. It has the disadvantage of not having the ability to scale on demand.
- **Community Cloud** - It is shared by organizations with similar interests, supported by a specific community, sharing the same mission or security requirements, etc.
- **Public Cloud** - It is available to the public or to a group of a big company. It is a multi-tenant cloud solution owned by a cloud service provider, which delivers shared hardware and software to costumer private networks (mostly the Internet) and data centers.
- **Hybrid Cloud** - Composed by two or more services (private, community or public), put together by standard or proprietary technologies, which allows portability. It takes advantages from the best of private and public models. Example: A client can implement a private cloud for applications with sensitive data and a public cloud for other, non-sensitive data.

Four levels of Cloud Computing Service Models:

- **Infrastructure-as-a-Service** - As the name suggests, it provides a computing infrastructure, such as virtual machines, firewalls, load balancers, IP addresses, virtual local area networks and others. Examples: *Amazon EC2*, *Windows Azure*.
- **Platform-as-a-Service** - Provides a computing platform which usually includes operating system, programming language execution environment, database, web server and others. Examples: *AWS Elastic Beanstalk*, *Windows Azure*, *Heroku*.
- **Software-as-a-Service** - Provides access to application software often referred as *on-demand self-service* software. Used without install, setup or run the application. Examples: *Google Apps*, *Microsoft Office 365*.

- **Business-Process-as-a-Service** - This model supply an entire horizontal or vertical business process and builds on top of any of services previously described.

In figure 2, it is possible to verify the differences between the several models.

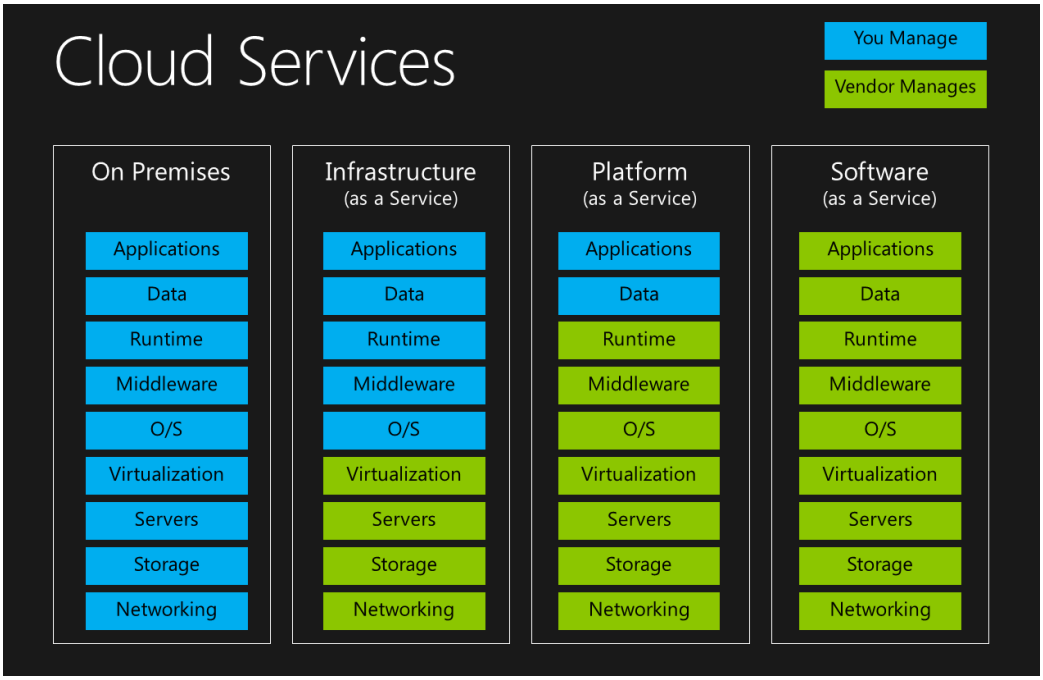

Figure 2: *Cloud computing service models. Source: www.hanusoftware.com/*

Nevertheless, such as any computer system, cloud computing is not free of external disturbances[1], the most important being:

- **Security attacks** - any attempt to gain unauthorized access;
- **Accidents** - an unplanned incident, resulting in damage;
- **Power surges** - an interruption of the flow of electricity;
- **Malfunction** - bugs cause bad functioning wrongly, or no functioning at all;
- **Worms** - malware computer program;
- **Distributed Denial of Service attacks** - an attempt to make an network resource unavailable.

3.2 Tools

In the initial phase beginning of (planning the basic software without any kind of user interface), it was necessary to research the best applications available, as the best way to obtain the planned results (a software fault injector).

For that reason, I thought that I could use the same tools used in Compilers course unit, Lex and Yacc or use others like Eclipse CDT, GCC Parser or MCPP preprocessor.

3.2.1 Bison/Yacc

Yacc is a parser generator and Bison is a GNU version of Yacc. Yacc picks up the tokens and builds a tree from it to check the syntax of the program. The lexer builds the tokens and they are declared in the yacc specification file. To use this tool it would be necessary to define the tokens and the grammar of the C language which would be laborious and time consuming, and that is not the focus of this internship.

3.2.2 Eclipse CDT

Eclipse CDT, as the name suggests, is a plugin for Eclipse that give a fully functional C and C++ Integrated Development Environment. Some of the features included in this plugin that are relevant for this project include:

- Source navigation;
- Code editor with syntax highlighting;
- Source code refactoring and code generation.

It is possible to use this plugin in standalone mode, by importing the .jar files to the project. Using it, I can code the fault injector in the java language, making it more maintainable and easy to use, write, compile and debug.

3.2.3 GCC Parser

Nowadays, GCC uses a hand-written parser to improve syntactic error diagnostics. This way, it is possible to provide meaningful messages on syntax errors to the users. Nevertheless, to use this parser in the injector, the learning curve would be very high and it would take a long time, since it is much optimized.

3.2.4 MCPP

MCPP is a portable C and C++ preprocessor with many features related with validation. Robert Natella used it as a workaround for some of the shortcomings of the GCC's C preprocessor. Now, it is outdated, since the last update was on 28-05-2013.

In the end, I chose to use the Eclipse CDT Plugin in standalone mode (only importing the necessary libraries to the project), because of my Java programming skills, the maintainability of the software developed in it and the low learning curve that the developers need to modify it.

4 Fault injector development

The Fault Injector currently in development is coded in Java using the Eclipse CDT Plugin, and it will implement thirteen operators (which can be seen in table 2)[13]. Furthermore, the fault injector can have two different schemas to trigger the faults: spatial and temporal. In the temporal way, the insertion of the fault is given by the time associated with the execution in the system, whereas, in the spatial way, the fault is injected when it reaches the specified zone where the particular operator can be applied.

Fault Type	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 and 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

Table 2: *Fault emulation operators.*

In the beginning of this project, I considered all the eighteen operators more representative in the open source software model. However, after collecting information and analyze them, I verified that I do not have all the necessary information to implement all the eighteen operators due to various reasons. The operators that will not be implemented can be seen in the table 3.

Fault Type	Description
EVAV	Extraneous variable assignment using another variable
MFCT	Missing functionality
WALL	Wrong algorithm - large modifications
WLEC	Wrong logical expression used as branch condition
WSUT	Wrong data types or conversion used

Table 3: *Other fault emulation operators.*

The reasons for which they will not be implemented are:

- Production of a large number of mutations;

- Inconclusive definition of the operators;
- Little or no information about the cases where they are applied;
- Can produce warnings or even errors while compiling;
- Low representation in relation to other operators (that can be seen in João Durães' field study).

To overtake some of these factors, it was necessary to obtain the data with which João Durães performed his field study, or do a new one. However, due to time constraints, it would be unfeasible to do it at this time.

4.1 Generate derivations

I chose to use a set of the most representative faults, previously specified by João Durães[7] according to his data-field results, specified individually below:

4.1.1 MFC

- Missing function call
The emulation of this operator is based in the removal of a function call in a context where the returned value is not used. Nevertheless, to do this removal, the constraints below need to be validated.

- **C01** - Return value of the function must **not** be used;
- **C02** - Call must **not be** the only statement in the block.

4.1.2 MIA

- Missing if construct around statements - **Implemented**
This operator simulates a missing *if* condition surrounding a set of statements. This causes the statements to be always executed and not only when the condition of the *if* statement is true.

- **C08** - The if construct must **not be** associated to an else construct;
- **C09** - Statements must **not include** more than five statements and not include loops.

4.1.3 MIEB

- Missing if construct plus statements plus else before statements - **Implemented**
This operator generates derivations of the source code of applications by removing the if construct plus statements plus else before statements. To apply this operator the following constraint must be verified first:

- **C08n** - The if construct must **be** associated to an else construct.

This constraint does not exist in João Durães specification, but as this operator cannot be applied in all situations, I specified and implemented it.

4.1.4 MIFS

- Missing if construct and surrounded statements - **Implemented**
The application of this operator changes the source code with the removal of one *if* construct and the statements surrounded by it. To do that, I need to verify the following constraints:

- **C02** - Call must **not be** the only statement in the block;
- **C08** - The if construct must **not be** associated to an else construct;
- **C09** - Statements must **not include** more than five statements and not include loops.

4.1.5 MLAC

- Missing and sub-expr. in logical expression used in branch condition - **Implemented**

This operator emulates the removal of part of a logical expression used in a branch condition. To apply this operator, the code must have at least two branch conditions linked together with the logical operator AND. With an AND operator, if one of the sub-expressions is *false* all the expression will be *false* and the condition will fail.

- **C12** - Must have **at least two** branch conditions.

4.1.6 MLOC

- Missing or sub-expr. in logical expression used in branch condition - **Implemented**

This operator emulates the removal of part of a logical expression used in a branch condition. To apply this operator, the code must have at least two branch conditions linked together with the logical operator OR. It is only necessary that one of the sub-expressions is true to the entire expression be evaluated as true. This operator has only one constraint:

- **C12** - Must have **at least two** branch conditions.

4.1.7 MLPA

- Missing localized part of the algorithm

As the name suggests, this operator emulates the omission of a small and localized part of the algorithm.

- **C02** - Call must **not be** the only statement in the block;
- **C10** - Statements are in the same block, **do not include** more than five statements, or loops.

The constraint **C02** guarantees that not all the statements in a block are removed, because this would not correspond to a realistic fault. This type of faults never involved the removal of *if* or *if-else* and loop constructs (the omitted statements were always function calls and assignments) guaranteed by constraint **C10**.

4.1.8 MVAE

- Missing variable assignment with an expression

This operator reproduces the omission of a given local variable with an expression.

However, constraint **C07** ensures that this does not happen when it is the first assignment to a variable, i.e. an initialization.

- **C02** - Call must **not be** the only statement in the block;
- **C03** - Variable must **be** inside stack frame;
- **C06** - Assignment must **not be** part of a for construct;
- **C07** - Must **not be** the first assignment for that variable in the module.

4.1.9 MVAV

- Missing variable assignment with a value

Operator **MVAV** is similar to operator **MVAE**, with the difference that it emulates the removal of the assignment of a given local variable with a constant value instead of an expression. The constraints related with this operator are the same of **MVAE**:

- **C02** - Call must **not be** the only statement in the block;
- **C03** - Variable must **be** inside stack frame;
- **C06** - Assignment must **not be** part of a for construct;
- **C07** - Must **not be** the first assignment for that variable in the module.

4.1.10 MVIV

- Missing variable initialization with a value

As the name suggests, this operator represents the removal of a given local variable initialization with a constant value. The fact that this operator only searches for variable initialization induce that only the first occurrence of an assignment to a particular variable are eligible to apply this type of fault. This is guaranteed by constraint **C04**. The constraint **C05** verifies if the assignment does not occur inside a loop, because one assignment of this type occurs several times. Nevertheless, this operator has other associated constraints:

- **C02** - Call must **not be** the only statement in the block;
- **C03** - Variable must **be** inside stack frame;
- **C04** - Must **be** the first assignment for that variable in the module;

- **C05** - Assignment must **not be** inside a loop;
- **C06** - Assignment must **not be** part of a for construct.

4.1.11 WAEP

- Wrong arithmetic expression in parameters of function call

This operator represents the modification of the expression used as parameter of a function call.

4.1.12 WPFV

- Wrong variable used in parameter of function call

Operator WPFV, as the name suggests, modifies the variables used as parameters in a function call, given a wrong variable. The use of constraint **C11** guarantees that there must be at least two variables in the module.

- **C03** - Variable must **be** inside stack frame;
- **C11** - There must **be at least** two variables in this module.

4.1.13 WVAV

- Wrong value assigned to a variable

As the name suggests, this operator simulates an assignment of a wrong value to a variable. This value is obtained by the inversion of bits of the least significant byte of the early value. To do that, the operator needs to verify the following constraints:

- **C03** - Variable must **be** inside stack frame;
- **C04** - Must **be** the first assignment for that variable in the module;
- **C06** - Assignment must **not be** part of a for construct.

The above operators may change, since they were specified for implementation at the binary level and in this project, they will be implemented at the source code level. After applying the operators in the code tree, modified files to use in the testing process.

4.2 Constraints

As it was discussed during the specification of the operators, the operators cannot be applied in all the situations and need to comply with the specification in accordance with the field-data study, which has been done by João Durães. To do that, the operators need to verify the constraints below.

Constraints	Description
C01	Return value of the function must **not** be used
C02	Call must **not be** the only statement in the block
C03	Variable must **be** inside stack frame
C04	Must **be** the first assignment for that variable in the module
C05	Assignment must **not be** inside a loop
C06	Assignment must **not be** part of a for construct
C07	Must **not be** the first assignment for that variable in the module
C08	The if construct must **not be** associated to an else construct
C09	Statements must **not include** more than five statements and not include loops
C10	Statements are in the same block, **do not include** more than five statements, or loops
C11	There must **be at least** two variables in this module

Table 4: *Fault emulation contraints defined by João Durães.*

The constraint **C07** is similar to constraint **C04**, one is the negation of another. This happens too with the constraint **C08** and **C08n**. Constraint **c10** is the same as constraint **C09**, but with one additional restriction: the statements need to be contiguous and need to belong to the same code block.

When operators **MIEB** and **MLOC** were being implemented, it was necessary to define the constraints **C08n** and **C12**. The constraint **C08n** was created because of the operator **MIEB** cannot be applied to an *if* construct without an *else* construct and the constraint **C12** was created because the operator **MLOC** cannot be emulated in a branch with only one condition.

Constraints	Description
C08n	The if construct must **be** associated to an else construct
C12	Must have **at least two** branch conditions

Table 5: *Other constraints.*

These constraints can be modified during the implementation of the other operators that are not implemented yet.

5 Work and implications

In figure 3, it can be seen an overview of the main decisions taken by me personally during this semester.

Since the beginning of this project, I was expected to implement a fault injector. However, as stated earlier, in the beginning it was supposed to inject hardware faults, but due to the postponement of six months, and the development of the project related to hardware faults injection, the project was then modified to inject software faults.

After taking that decision, I had to choose the technique that I would use to inject faults, from three different ones: at binary code level in the execution environment, at object code level in the compilation or before the compilation at source code. I have selected the source code level technique because there are some available tools, which inject faults in execution environment, e.g. by João Durães. Previously, Robert Natella had coded a similar tool, using MCPP as a C preprocessor during his PhD thesis. All these factors would lead to choosing to inject faults at object code level in the compilation, but the use of this technique in the development of this fault injector and the evaluation of the robustness of the cloud would be a great effort and too much work for a master's thesis. Hence, it was decided to inject faults before compilation time, at the source code of applications. The use of this technique provides the emulation of realistic software faults done by real programmers.

Despite the existence of Robert's tool to inject faults at source code, the injector under development will use the capabilities of Java, such as the maintainability and its easy of use, to inject faults in source code coded in the C language.

The faults will be injected in C code because of the extensive knowledge of the supervisors with this programming language and also because of the work already done by João Durães during his PhD in the specification of the operators, which was based on a field-study of open source software coded in that language.

Then, I evaluated the software possibilities to parse the code and get the AST tree. I chose to use the Eclipse CDT Plugin mainly because of my abilities in programming in Java Language, but also by the features that it has, such as source navigation, source code refactoring and code generation.

Despite having many attractive characteristics for this project, the implementation of the first operator with Eclipse CDT was not easy. After some time spent to understand the tool and its class structure of classes through Javadoc, it was even necessary to obtain more information from those who know and really work with it, by accessing the official mailing list: cdt-dev@eclipse.org.

After exchanging some emails with Thomas Corbat, I learned that Eclipse CDT does not allow the creation of a new code tree by changing the original tree. Moreover, I had two options, use reflection to get the modifications from ASTWriter and pass it to ASTRewrite to get the code with the changes done or get the source code of CDT and change it to avoid using reflection.

Initially, I opted to use reflection, despite knowing that it is not a neat solution and is generally slower than equivalent native code, but after understanding

better the flow of Eclipse CDT, I got the source code and changed it to avoid the use of reflection.

After that, I finally had the first operator implemented, using recursion. However, I then realized I can traverse the tree without using recursion, by using the Visitor Pattern, making the code simpler, cleaner and safer. Then I modified all the recursion to the visitor pattern.

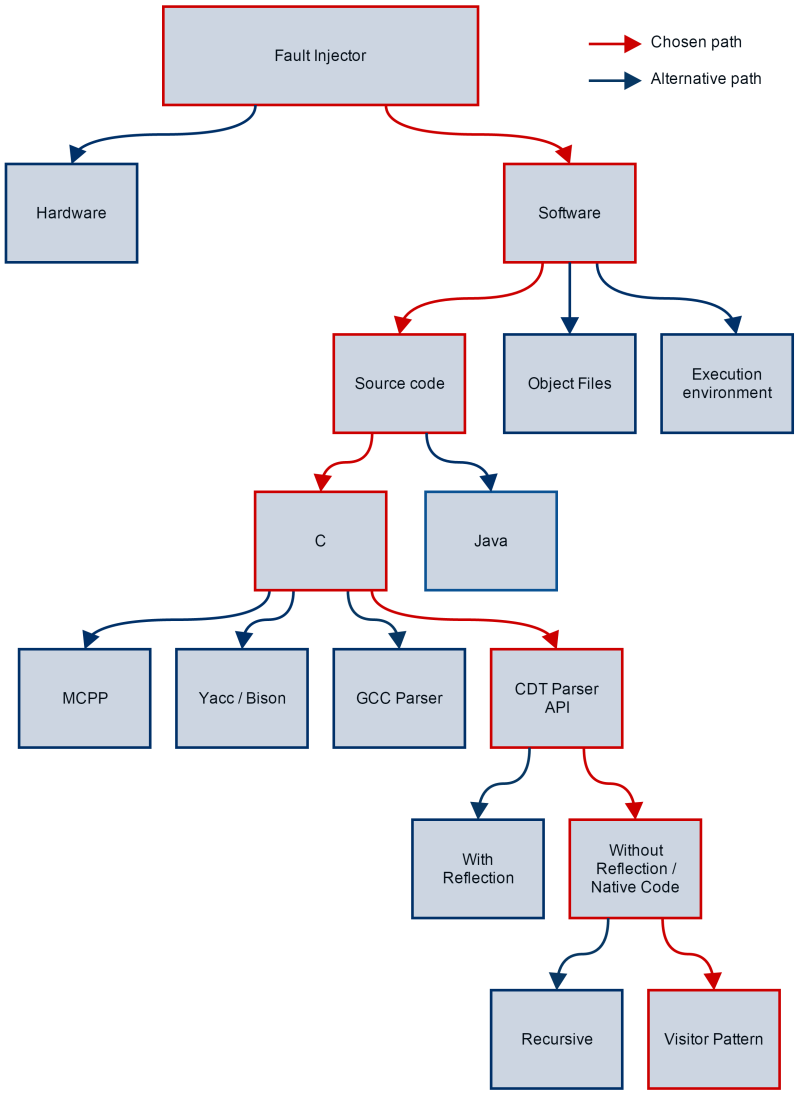

Figure 3: *Decision tree.*

Figure 4, represents an overview of the fault injection tool. Initially, the fault injector starts by reading the source code of a file coded in C. The code is then analyzed and a AST tree is then created. To inject a fault, the fault injector finds the node where it can be injected, and modifies it, according to operator specification. After this, the AST tree is rewritten, getting the code again, now with modifications. Finally, with the comparison of the two source codes, source code and source code with mutations, it is made a *diff*, so obtain a summary of the changes made between files.

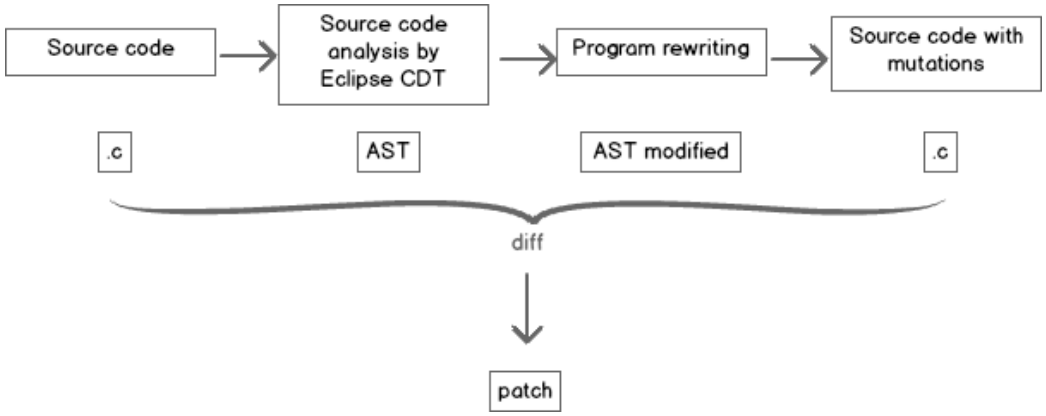

Figure 4: *Overview of the injection tool.*

I chose to use the *diff* tool, since the use of this tool allows the creation of smaller files with only the changes that are made, instead of using all the code with the modifications made in it.

6 Conclusion

6.1 Global vision

In table 6, it is possible to view (marked green), the operators that were implemented in the first semester of this dissertation. As it can be seen, I have implemented five of the thirteen operators that João Durães specified. The first operator that I have implemented with success was the MIFS one, and as the operators MIA and MIEB are similar and have some constraints in common, I also implemented them.

Fault Type	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 and 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

Table 6: *State of the operators.*

In table 7, is also possible to check that I have implemented three of the eleven constraints related to the thirteen operators, represented in green.

Constraints	Description
C01	Return value of the function must **not** be used
C02	Call must **not be** the only statement in the block
C03	Variable must **be** inside stack frame
C04	Must **be** the first assignment for that variable in the module
C05	Assignment must **not be** inside a loop
C06	Assignment must **not be** part of a for construct
C07	Must **not be** the first assignment for that variable in the module
C08	The if construct must **not be** associated to an else construct
C09	Statements must **not include** more than five statements and not include loops
C10	Statements are in the same block, **do not include** more than five statements, or loops
C11	There must **be at least** two variables in this module

Table 7: *State of the constraints.*

Constraints	Description
C08n	The if construct must **be** associated to an else construct
C12	Must have **at least two** branch conditions

Table 8: *State of the other constraints.*

Below, in figure 5, it can be seen the version numbering system of this fault injector. The "b" letter states that there are two constraints implemented which are not specified by João Durães. For instance, if I had implement three constraints, then it will be "c", and so on. The current version of the injector also has five operators and three constraints implemented, from those specified by João Durães. When the version numbering reaches 0.13.11 then the injector will be in version number one.

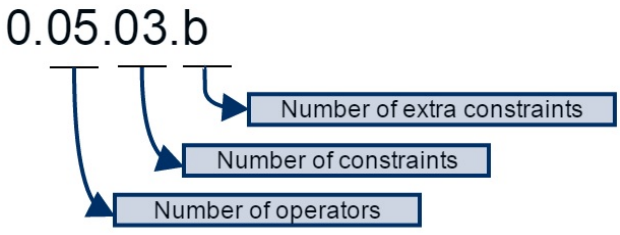

Figure 5: *Version number.*

6.2 Future work and experiments

In the future, I have planned to implement the remaining operators and constraints. I will use regression testing to verify that when I code one new operator or constraint I do not mess with the operators and constraints previously implemented. In addition to this, I will add to the fault injector a user interface, in order to make it more user-friendly, and a name.

Furthermore, in the next semester, I will study the hypervisor and I will implement some scenarios to evaluate the behavior of the cloud with and without faults.

In figure 6, it can be seen two environments where the first and second experiments will be done, under normal conditions (without faults) and with faults, respectively. An application will be selected, as shown in the figure as "App". This application will run in a normal environment and it will be measured the runtime and the result of the execution, for later comparison with the results obtained in other scenarios.

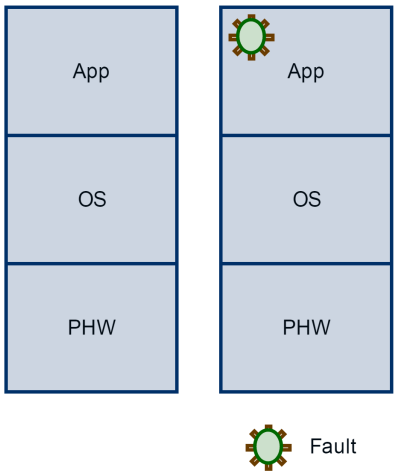

Figure 6: *First and second experiments.*

The scenario represented in the right side of figure 6 is similar to the one in the left side, with the slight difference that the latter one has a fault injected into the "App". Depending on the type of fault injected into the "App", it will have different behaviors, which will be assessed by the *CRASH Scale*.

Below, in figure 7, it is represented the third experiment with an Native Hypervisor. The goal of this scenario is to evaluate whether through fault injection in one of the virtual machines, the others are affected. It will also try to evaluate if the virtual machine without faults exhibits the same behavior when running side by side with a virtual machine that has an "App" with faults and without faults.

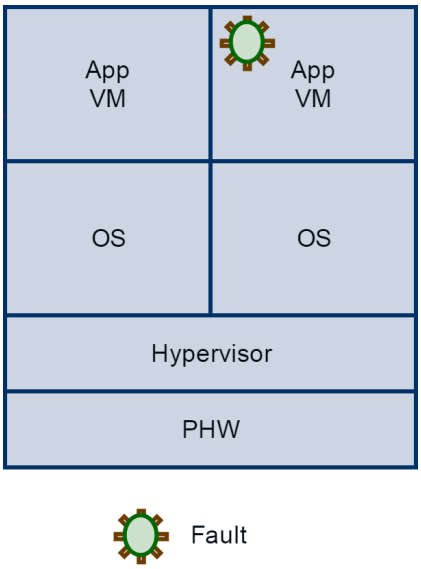

Figure 7: *Third experiment.*

A Gantt diagrams

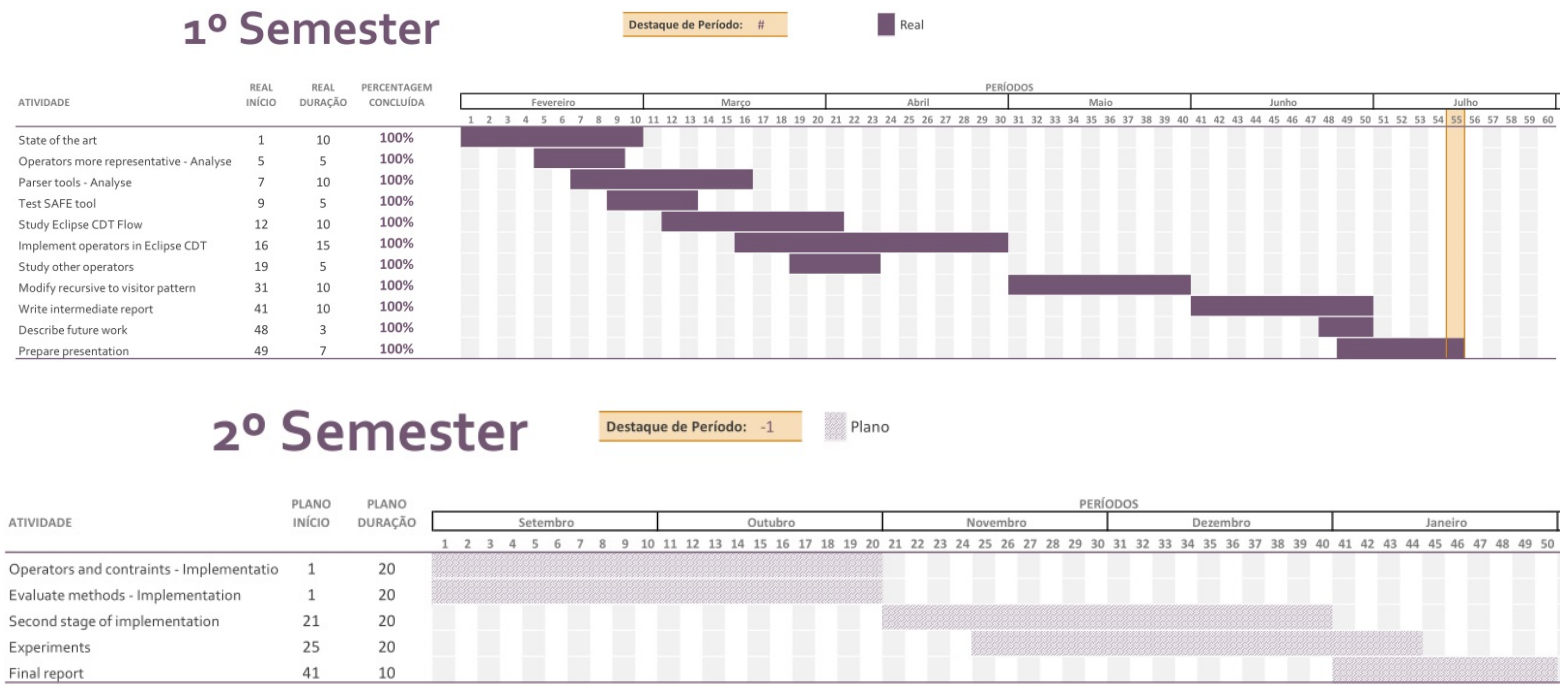

Figure 8: *First and second semester Gantt.*

B Risks table

Risc Area	Preventative Measures	Recovery Measures
Equipment Failure	Ensure regular maintenance is undertaken	Use alternative sources/type of equipment as appropriate
	Allow for sufficient funding for repairs	
	Indentify alternative sources/type of equipment	
Data lost	Back-up data regularly	
Publication of similar research	Regularly search electronic publications databases	Modify project
	Continue literature review throughout candidature	
	Ensure timely submission	
Personal issues interfere with progress	Take leave of absence (unless for sickness or bereavement)	Re-apply for admission when able to commit
	Take annual leave	
	Take sick leave	
	Communicate with supervisor	
Student loses interest	Select motivating topic at the start	
	Enrolling area ensures a dynamic research culture	
	Improve communication between student and supervisor	
	Look for warning signs	
	Register for support programs/seminars	
	Talk to fellow students in research area	
Dispute between student and supervisor	Understand each other's roles and expectations	
	Agree on dispute resolution process when initiating relationship	
Supervisor takes excessive time to check final drafts	Supervisor to plan out workload	
	Student plan ahead to ensure supervisor will be available	
	Student/Supervisor to review chapters/sections at regular intervals	
Student wants to submit thesis without supervisor approval	Student to be counselled regarding implications - a recomendation of fail or major revision from examiners likely if thesis below standard	Review of thesis by alternative person within University recommended

Figure 9: *Risks.*

References

[1] K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, *Resilience assessment and evaluation of computing systems.* Springer, 2012.

[2] A. Avizzienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing."

[3] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on.* IEEE, 2000, pp. 417–426.

[4] L. Regina, E. Martins *et al.*, "Jaca—a software fault injection tool," in *null.* IEEE, 2003, p. 667.

[5] E. Martins, C. M. Rubira, and N. G. Leme, "Jaca: A reflective fault injection tool based on patterns," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on.* IEEE, 2002, pp. 483–487.

[6] B. P. Sanches, T. Basso, and R. Moraes, "J-swfit: A java software fault injection tool," in *Dependable Computing (LADC), 2011 5th Latin-American Symposium on.* IEEE, 2011, pp. 106–115.

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

[8] N. Bridge and C. Miller, "Orthogonal defect classification using defect data to improve software development," *Software Quality*, vol. 3, no. 1, pp. 1–8, 1998.

[9] R. Chillarege, *Orthogonal Defect Classification.* Handbook of Software Reliability Engineering, ed. Michael R. Lyu (Los Alamitos, CA: IEEE Computer Science Press, 2004.

[10] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on.* IEEE, 1997, pp. 72–79.

[11] P. Mell and T. Grance, "The nist definition of cloud computing," 2011.

[12] E. Schouten, *IBM® SmartCloud® Essentials.* Packt Publishing Ltd, 2013.

[13] J. A. Duraes, "Faultloads baseadas em falhas de software para testes padronizados de confiabilidade," *Thesis*, pp. 0–269, 2005.