Master in Informatics Engineering
Dissertation
Thesis Report

Evaluating the robustness of the Cloud

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

Supervisor:
Raul Barbosa

Co-Supervisor:
Henrique Madeira

Friday 29th January, 2016

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

Master in Informatics Engineering
Dissertation
Thesis Report

Evaluating the robustness of the Cloud

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

Supervisor:
Raul Barbosa

Co-Supervisor:
Henrique Madeira

Arguing member:
Filipe Araujo

Vowel:
Alexandre Miguel Pinto

Friday 29th January, 2016

To all my family, my girlfriend and my friends.

Acknowledgements

This work was conducted under the guidance of professors Raul Barbosa and Henrique Madeira to whom I would like to express my sincere gratefulness for their valuable support.

I am very grateful to my girlfriend Andreia for her encouragement and patience. I would also like to thank my friends and colleagues of Informatics Engineering Department for all those times they have given me support.

Last but not least, I would like to express my thankfulness to my family in Portuguese. *Pelo vosso apoio incondicional, amor, compreensão e disponibilidade, o meu mais verdadeiro agradecimento. Obrigado!*

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

Albert Einstein

Resumo

Cloud Computing é um conceito que permite aos utilizadores tomar partido da tecnologia e, ao mesmo tempo, concentrarem-se no seu *core business*. Com este conceito, é possível assegurar o funcionamento das tecnologias envolvidas, abstraindo-se de várias preocupações e dificuldades tecnológicas, como, por exemplo, a manutenção dos equipamentos. O uso de *Cloud Computing* oferece diversas vantagens como acesso alargado à rede, associação de recursos, recursos à medida, elasticidade, escalabilidade e um serviço à medida através de quatro modelos diferentes (*community*, híbrido, privado e público). No entanto, como a *Cloud* não é livre de perturbações, como ameaças à segurança, falhas de energia, sobrecargas de trabalho e falhas de *hardware* e *software*, é pertinente avaliar até que ponto é que a mesma é tolerante a falhas, neste caso, falhas de *software*. Portanto, a presente dissertação aborda o desenvolvimento de uma ferramenta de injeção de falhas, denominada BugTor. Para avaliar a resposta do sistema às ditas perturbações, esta ferramenta introduz erros e defeitos no *software*. Este injetor tem como principal característica a injeção de falhas diretamente em código fonte. O principal contributo desta dissertação para a continuidade da investigação relativamente à avaliação da robustez tanto da *Cloud*, vai ser a criação de um software de injeção de falhas, e de *scripts* auxiliares de forma a avaliar e validar o mesmo, assim como mecanismos de avaliação dos resultados da injeção de falhas no servidor Web Apache.

Palavras-chave: *Cloud Computing*, Erros, Falhas, Injecção de Falhas, Robustez, Tolerância a Falhas, Vulnerabilidades.

Abstract

Cloud Computing is a paradigm that allows users to take advantage from the technology and, at the same time, focus on their core business. Rather than being blocked due to technological difficulties, users can get the most of the technology without having knowledge or skills to ensure the proper functioning of all the technologies involved. The use of Cloud Computing provides advantages such as broad network access, resource pooling, on demand self-service resources, rapid elasticity and a measured service through four different models (community, hybrid, private and public). However, Cloud Computing is not free of external disturbance such as security attacks, power surges, workload faults, hardware and software faults. For that reason, it is pertinent to assess its behavior in the presence of software faults. Therefore, the current dissertation addresses the development of fault injector software, named BugTor. In order to evaluate such disturbances, this tool introduces errors and defects into the software in several test cases. This injector is based on the injection of software faults at source code level. The main contribution of this work for the continuation of research in this area, is the creation of a fault injection software, and some scripts. The purpose of these scripts is to evaluate and validate the fault injector and to collect information about the results of fault injection at Apache.

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

Contents

List of Figures

List of Tables

Listings

Abbreviations

API Application Programming Interface

APXS APache eXtenSion tool

AST Abstract syntax tree

BPaaS Business-Process-as-a-Service

CISUC Center for Informatics and Systems of the University of Coimbra

EMP Electromagnetic pulse

G-SWFIT Generic Software Fault Injection Technique

HWIFI Hardware Implemented Fault Injection

IaaS Infrastructure-as-a-Service

IT Information Technology

NOC Network operations center

ODC Orthogonal Defect Classification

PaaS Platform-as-a-Service

SaaS Software-as-a-Service

SFI Software Fault Injection

SSE Software and Systems Engineering

SWIFI Software Implemented Fault Injection

V&V Verification and Validation

VVM Virtual Machine Monitor

Operators

EVAV Extraneous variable assignment using another variable

MFC Missing function call

MFCT Missing functionality

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

WAEP Wrong arithmetic expression in parameters of function call

WALL Wrong algorithm - large modifications

WLEC Wrong logical expression used as branch condition

WPFV Wrong variable used in parameter of function call

WSUT Wrong data types or conversion used

WVAV Wrong value assigned to a variable

Constraints

C01 Return value of the function must **not** be used

C02 Call/Assignment/The if construct/The statements 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, nor loops

C11 There must **be at least** two variables in this module

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

C13 The if construct must **be** associated to an else construct

1 Introduction

The present dissertation describes the work developed in the scope of Master's degree in Informatics Engineering of the University of Coimbra and took place in CISUC (Center for Informatics and Systems of the University of Coimbra), particularly in the Software and Systems Engineering (SSE) Research Group.

Throughout this work, the robust of a cloud platform is assessed. Cloud Computing denominates not only applications delivered as services through the Internet, but also the software and hardware used in data centers to provide these services [Armbrust et al., 2010]. Also known as "the cloud" or "cloud", Cloud Computing is achieved when the components such as data storage, software and services are extracted to remote data centers. Following this way, cloud users can focus on their core business, without the inconvenience of spending time and resources to ensure the proper function of all involved components.

In cloud conditions, despite its numerous benefits, applications are threatened by several types of failures, such as network, hardware and software failures, affecting their reliability, which means impacting the ability to perform at a specific level. The Institute of Electrical and Electronics Engineers (IEEE) [ISO/IEC/IEEE 24765, 2010] defines reliability for software as "capability of the software product to maintain a specified level of performance when used under specified condition".

Regarding software failures in data centers structures where applications are executed [Ahmed, 2009], they are mostly caused by:

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

According to Avižienis [Avižienis et al., 2004], the failures happen when the error lead the provided service to deviate from correct service. However, it was required to specify narrowed categories:

- **Error**: Part of the total state of the system that may lead to a service failure;
- **Failure**: Incident in which the function of a system is not performed within the functional specification.
- **Fault**: The cause of the error in a software product.

Software quality assurance is a demanding task, considering that software bugs and errors are a systematic part of computer applications [Nindel-Edwards and Steinke, 2014]. If an organization requests a service from a cloud provider (for example from Microsoft Azure [1], Amazon EC2 [2] or Google Cloud [3]), such as storage of an application, this organization should accept the policies given by the cloud application provider.

The number of organizations (both private and public) which use Internet services and Cloud Computing is increasing [Diez and Silva, 2014]. These technologies are used for almost all types of systems, including critical systems. According to Sommerville [Sommerville, 2006], a critical system "is any system whose failure can result in significant economic losses, physical damage or threats to human life".

Although there are solid virtualization platforms, fault tolerance is still a problem in research. Fault tolerance is related to assuring the delivery of a correct service, even in the event of system faults. The ability of system to recover from the failures existence, named resilience, is a critical factor in the cloud [Jhawar et al., 2012]. In the cloud, the resilience allows applications to be reliable, providing a specific service continuously.

Assessing cloud robustness is an important task. Robustness "is the ability of a software system to deliver service in conditions which are beyond its normal domain of operation" [De Florio, 2012].

Several projects have been performed previously in this area. Particularly within CISUC-SSE Research Group, it is relevant to refer the following projects:

- *Faultloads baseadas em falhas de software para testes padronizados de confiabilidade* [Durães, 2005];
- *Achieving representative faultloads in software fault injection* [Natella, 2011];
- *Benchmarking de Infraestruturas de Virtualização para a Cloud* [Cerveira, 2015].

In his thesis, Durães [Durães, 2005] looked into the software faults and its inclusion in the dependability benchmarks. A dependability benchmark is a procedure that aims to provide a standardized way to measure the dependability and performance of a system or components [Vieira and Madeira, 2009]. During his work, Durães faced a few technical issues, such as representativeness, characterization and injection techniques of software failures. Furthermore, Durães [Durães, 2005] performed a field study based in real software faults and proposed a classification schema for the injection faults. He classified all the faults using the Orthogonal Defect Classification Model and, after that, he grouped the faults by the nature of the defect in the programmer point of view (missing, wrong or extraneous construct). Finally, he specified the types according to the possible instantiation that each fault can take and the context where the fault was discovered.

Through this study, it was concluded that there is a group of faults which represents more than half of the total faults. It was possible to define faultload as a set of faults and stressful conditions that can emulate real faults, within the dependability benchmark [Durães, 2005].

Moreover, it was proposed a technique to emulate this type of faults through the modification of executable code. This technique does not require the source code, which makes it a good technique to dependability benchmark scenarios. Finally, he presents a set of operators to perform the emulation of these types of representative faults identified in his field-study.

During his PhD Thesis, Natella [Natella, 2011] approaches the problem of representativeness of faults, focusing on improve it, by selecting the most favorable fault-existence components. This allows increasing the representativeness of faults, reducing the faultload size and the cost associated with it. Furthermore, Natella improves the precision of techniques and tools that inject faults at binary level, such as Generic Software Fault Injection Technique (G-SWFIT). Finally, he researches the injection of concurrency faults and proposes a new technique, injecting faults at execution environment and triggering conditions that activate these faults through the thread management (thread scheduling and locking operations). This technique provides a closer approach to emulating the behavior of concurrency faults, than the other techniques.

The thesis of Cerveira [Cerveira, 2015] evaluates the resilience in the Cloud Computing systems, within the process of migration from in-house to the Cloud. Such process can raise doubts

and mistrusts among the organizations. Thus, organizations are forced to trust in external entities on which they do not have complete control. In order to evaluate the resilience of this type of systems, Cerveira [Cerveira, 2015] developed a watchdog mechanism to detect issues and restore the service of virtualized systems. At the same time, he also created the basis to achieve resilience benchmarks for virtualized systems.

1.1 Objectives

The main objective of this work is to contribute to the evaluation of the dependability and robustness of the cloud. In order to achieve this goal, a fault injection tool will be designed and implemented.

According to Cotroneo [Cotroneo, 2013], a fault injection tool allows the introduction of "faults in a system in order to assess its behavior and to measure the efficiency (i.e. coverage and latency) of fault tolerance mechanisms".

Particularly, the purpose of this tool is to inject software faults in the source code of specific software components.

Within the main objective, the following goals are determined:

- Implement the thirteen operators specified by Durães & Madeira [Durães and Madeira, 2006];
- Use the fault injector to emulate faults in applications and software components in general;
- Verify and measure the behavior of running the application, in normal conditions (typically know as Golden Run);
- Inject faults in the target application, verify and analyze the effects.
- Compare the results between the scenario with normal conditions and the scenario with faults;
- Create a scenario with multiple virtual machines, verify and analyze the effects. Measure the value and the time in which the application runs, with and without faults; Evaluate the results and conclude about the dependability and robustness of the cloud.

1.2 Document Structure

The second chapter presents the state of the art in related areas with particular emphasis in the software fault injectors and the applications used to get source analysis. Moreover, this chapter introduces the categories of ODC model, the characteristics of Software Fault Injection and Cloud Computing, as well as the differences between Hypervisor and most used WebServers. Finally, it describes two scales to evaluate the obtained results.

The third chapter describes the development of fault injector, named BugTor, focusing the operators that represent the most common software faults and the restrictions related with each operator. Then, it is explained the workflow and some particularities of the implementation. Afterwards, the requirements and usage method necessary to run the injector are presented. Lastly chapter three, the methodology applied to perform the validation is exposed, as well as its limitations.

The fourth chapter discusses the work done and its implications in the project; the research involved to perform this work and the decisions taken during this project based on both research results and knowledge.

The fifth chapter presents the results of the experiments performed during this project, with particular emphasis on the specification of the setup where experiments are executed, the classification of different results obtained and the results and its analysis.

In the last chapter, the conclusions are reported, as well as the work that should be done in the future.

1.3 Methodology

The adoption of a project management methodology is essential. It was given to the student the freedom of choosing the best methodology to cover both project objectives and his needs. The chosen methodology should allow a continuous improvement of the progress of the project. Also, the student should feel comfortable in its application.

Regarding the development of this project, an *Agile Life Cycle* based in an *Incremental Model* was used, without diminishing the ultimate objective.

The adoption of this model allows the following: reach quickly whenever the existing requirements are changed or new requirements are added during this project; better monitoring of the progress of the project, through the execution of successive assessments during the developed activities, which has the advantage of timely feedback to eventual adjustments.

1.3.1 Meetings

In order to track the progress of this work and make sure the proposed goals are achieved, weekly meetings were scheduled between the student and Dr. Raul Barbosa. In addition, in the first semester, the student attended a few general meetings of the project in which important concepts and the course of the project were discussed together with Dr. Raul Barbosa (supervisor), Dr. Henrique Madeira (co-supervisor), and two other colleagues with similar projects.

1.3.2 Risks

There are a few risks which can affect the development of this work. In order to overcome the risks related to equipment failure and data lost, *GitHub* was used to backup all the sources of the project and this report. These backups are done on a daily basis and every time a modification is performed.

Nowadays, several investigation centers are focused on studying the robustness of the cloud. Therefore, similar research can be published during the development of this project. With the purpose of avoiding this risk, the student will, regularly, verify the latest publications on this matter. If similar research is published, changes in the project will be addressed to make sure the performed work will add value and benefit Science and Research.

These risks, among others, together with the respective preventive and recovery measures are presented in Appendix B.

1.3.3 Planning and Tracking

A few tasks have been clearly assigned in the beginning of this work. Afterwards, these tasks were adapted considering that the student felt the need of postponing the beginning of the dissertation for about six months. Thus, the initial proposed Gantt presents a few differences compared with the final first semester Gantt, such as the change from hardware to software. The two Gantt diagrams of both semesters are presented in Appendix A, with the most important tasks performed, as well as their duration.

Regarding the work performed in the first semester, a few tasks were prioritized, such as:

- study of related works, as well as technologies and tools;
- analyses of most representative faults;
- the implementation of some operators that emulate software faults.

According with the final first semester Gantt, all the tasks have been completed and carried out according to the plan.

Regarding the second semester, all the phases of implementation previously planned were concluded, but with a longer duration than expected. However, the second phase of fault injector implementation occurred within the expected time, although it was overlapped with other tasks. At this stage of the work, there was the need of starting the experiments and writing the present report, before the planned timing. These tasks have been anticipated considering the importance of making experiments with fault injector in order to analyze the results as soon as possible.

In conclusion, all the main tasks have been performed, although some delay in the tasks of the second semester. The last tasks were anticipated, in order to obtain results on time. A few tasks were performed in a longer period of time than expected.

2 State of the Art

This chapter presents the techniques used to emulate software and hardware faults. Afterwards, Cloud Computing will be introduced, as well as its main characteristics, models of deployment and levels of service. Furthermore, several tools that help inject faults in software will be surveyed. In order to evaluate the robustness of the Cloud, it was necessary to simulate a Cloud environment through using a hypervisor. Therefore, the different types of hypervisors will be described in this chapter together with an analysis of the segment of webservers. Lastly, two approaches to evaluate the effects of emulated injections will be compared.

Nowadays, Cloud Computing Services are being used more and more by both individuals and organizations. The increasing use of the Cloud is justified by its numerous benefits, such as:

- Easy access to services at any time and from any location;
- Lower costs;
- Unlimited storage capacity;
- Scalability;
- Flexibility;
- Low usage of many dedicated servers;
- Reduction of noise margins;
- The cloud provider offers resources ready to deliver [Wolter et al., 2012].

Moreover, Cloud Computing Services are independent from the hardware. The above advantages motivate the organizations to migrate their applications and services to the cloud. In other words, several companies have their infrastructure mainly based on remote third-party cloud service providers.

Software and hardware faults are some of the disadvantages of the Cloud Computing, which makes resilience - the ability of the system to recover from failures - one of the critical factors. In fact, software quality assurance is a demanding task, considering that software faults and errors are a systematic part of computer applications [Nindel-Edwards and Steinke, 2014]. There are several studies showing that software faults (i.e. bugs) [Avižienis et al., 2004] are the main cause of computer failures.

Fault injection tools allows to study the effect of software faults in a system "in order to assess its behavior and to measure the efficiency (i.e. coverage and latency) of fault tolerance mechanisms", according to Cotroneo [Cotroneo, 2013]. The first developed injectors were implemented in hardware and targeted to the injection of hardware faults, thus these injectors are named tools Hardware Implemented Fault Injection (HWIFI) - Hardware Implemented Fault Injection.

This type of tools is mainly based in: the inversion of bits and placing of a specified value in some memory registers. Some examples of the application of this technique are Electromagnetic pulse (EMP) and radiation. However, these tools are limited by: the technology where they are implemented and the increasing of complexity of the circuits.

In order to overcome these limitations, injectors started being done using software, through the Software Implemented Fault Injection (SWIFI) - Software Implemented Fault Injection [Madeira et al., 2000]. The purpose of this technique is to emulate errors at the software level which happen during the execution environment, in both hardware and software.

This technique is able to inject errors/faults at different levels and in several target points. Some examples are:

- Data corruption in registers, memory or hard drive;
- Communication problems in network or Network operations center (NOC) (Network operations center);
- Software faults in binary code, in object files or in source code.

As presented in the Table 1, 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.

		Emulation environment	
		Software	**Hardware**
Fault injection environment	**Hardware**		HWIFI
	Software	SWIFI	SWIFI

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

SWIFI [Madeira et al., 2000] is an attractive technique because it does not require additional hardware, which would increase the cost of the test. However, it can not inject faults in inaccessible areas and it may disrupt or change the workload of testing software. The system main targets of this technique are the applications and operating systems. 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 [Madeira et al., 2000]. A technique, called G-SWFIT (Generic Software Fault Injection Technique [Durães and Madeira, 2006]), was developed from SWIFI. This technique was raised to modify the binary code of the target, reproducing the sequence of instructions that corresponds to the required fault injected in the source level.
- **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.

In order to create a software fault injector is necessary the use of one of the above techniques and is also required faults, or groups of them. Particularly, the purpose of the fault injection tool which will be developed in the current project is to inject software faults in the source code of some applications.

2.1 ODC Model

Besides the technique and the environment used to inject faults, the faults need to be classified. The Orthogonal Defect Classification [Bridge and Miller, 1998] Model is a framework with wide acceptance and widely used in the research community. It is developed by IBM [Chillarege, 2004] and created to improve the level of technology available to assist the decisions of a software engineer, by measurement and analysis.

In the field-study of Durães & Madeira [Durães and Madeira, 2006], they collected a large set of real software faults to analyze and take conclusions as consistent as possible. Durães started grouping the faults in the ODC model. The grouping of this framework is based in the changes that need to be done to correct a software fault which is characterized by eight categories:

- **Algorithm** - Problems that can be fixed by re-implementing an algorithm or local data structure, include efficiency or correctness that affects the task.

- **Assignment** - An assignment defect indicates an initialization of control blocks or a data structure.
- **Build/package/merge** - Errors that occur in the integration of library systems, management of changes, or in version control.
- **Checking** - Based on the program logic that is checked and fails to validate data and values before the usage, loop conditions, etc.
- **Documentation** - Errors in the documentation can disseminate to publications and maintenance notes.
- **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.
- **Interface** - Problems in the interaction with other components, modules, device drivers, call statements, control blocks, or parameter lists.
- **Timing/serialization** - Errors that happen in shared and real-time resources.

Durães [Durães, 2005] used this model in his field-study, as a starting point for fault classification. The categories Build/package/merge, Timing/serialization and Documentation were excluded due to: the inexistence of information at the time of the study and the inclusion not relevant for the study.

This categorization provides useful information for fault injection purposes and is useful for software reliability.

2.2 Injection of software faults

Software fault injection (SFI) is a testing technique used to understand how software behaves when stressed in unusual conditions to increase the levels of dependability of the application under test. The SFI has the following purposes:

- Find defects in software;
- Robustness testing;
- Determine failure modes;
- Safety verification;
- Security assessment;
- Software testability analysis;
- Used with or without source code.

Fault injection is an area that has fascinated researchers, mainly due to the need of assuring software quality, which is a tough challenge. A recent survey on software fault injection can be found in [Madeira et al., 2015]. In the context of the present thesis, it is relevant to present and compare some fault injection tools have contributed to advanced in this area of research and are directly related to the goals of the thesis. This includes the following tools:

- JACA [Regina et al., 2003];
- J-SWFIT [Sanches et al., 2011];
- SAFE [Natella, 2011];
- Xception [Carreira et al., 1998].

JACA [Regina et al., 2003] is a source-code independent tool which 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 at bytecode level [Martins et al., 2002]. The goal of this tool is to use high-level programming features to corrupt attribute values, methods parameters or return values during runtime.

On the other hand, Java Software Fault Injection Tool [Sanches et al., 2011] does not need the source code to perform the injection because the mutation of the code is performed directly in compiled code and it is based on the G-SWFIT [Durães and Madeira, 2006].

Regarding the SAFE [Natella, 2011] is an application which uses SWIFI technique 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 shortcomings of the GCC's C preprocessor. After that, some variations of original files are written (code with simple mutations) with the applied operators. Natella [Natella, 2011] implemented thirteen operators in SAFE, the same number as Durães & Madeira [Durães and Madeira, 2006]. However, Natella implemented them in the source code level, while Durães & Madeira in binary level.

Lastly, Xception [Carreira et al., 1998] is an automated testing tool which injects faults with minimum interference in the system workload, taking advantage from both the advanced debugging and performance monitoring features presented in the latest processors. Xception [Carreira et al., 1998] presents the following advantages:

- Reduction of the interference with the workload;
- Injection of quite realistic faults;
- Monitor the activation of the faults;
- Record detailed information regarding the injection.

Furthermore, Xception [Carreira et al., 1998] allows to inject faults in applications even when the source code is not available.

The fault injector under development in this thesis will be similar to SAFE [Natella, 2011] in terms of output: source code files with changes made in it. However, there is the need to create this injector considering that: the code developed by Natella [Natella, 2011] is not available and it is intended to design a more maintainable and easy to use injector, using Java Language without involving the MCPP preprocessor, which is already outdated.

2.3 Cloud Computing

In the current project, the injection of failures focuses in the cloud, including software commonly used in this environment. According to Mell and Grance [Mell and Grance, 2011]:

"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.".

Cloud Computing, represented in Figure 1, is a new way to deliver on-demand Information Technology (IT) services (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. Moreover, the use of this model enable an organization to narrow the fixed costs of having an IT infrastructure as well as allows the system to accommodate larger loads just by adding resources.

Below, Cloud Computing is described in relation to its characteristics, deployment models and service models [Schouten, 2013]. 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).

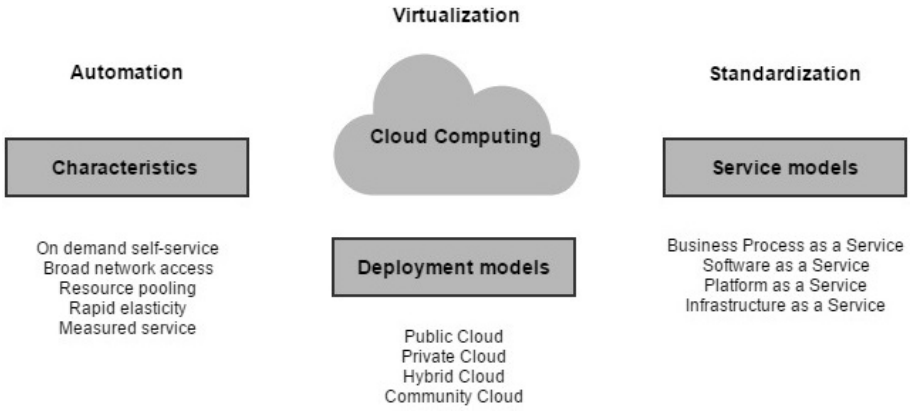

Figure 1: *Cloud computing overview.*

- **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 customers pay in accordance with the service spent.

The Cloud can be categorized in four deployment models:

- **Private Cloud** - It is a single-tenant cloud solution using client hardware and software, 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.

Besides the four deployment models, it should be also considered the 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, webserver 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. It is used without install, setup or run the application. Examples: *Google Apps*, *Microsoft Office 365*.

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

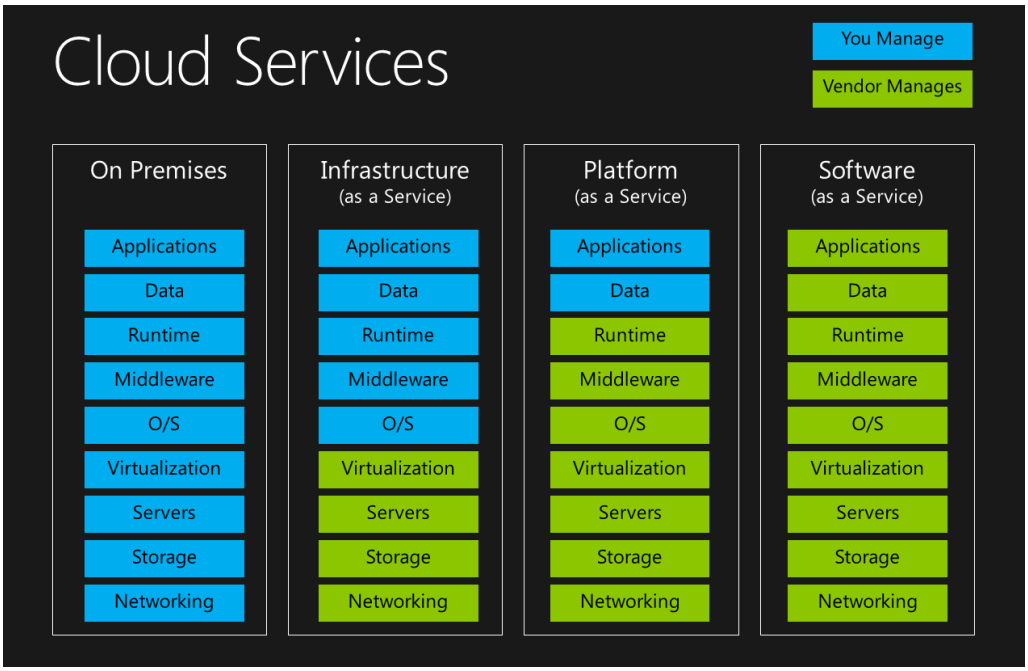

Figure 2: *Cloud computing service models. [4]*

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

However, such as any computer system, cloud computing is not free of external disturbances [Wolter et al., 2012], like:

- **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 or no function at all;
- **Worms** - malware computer program.

The fault injection tool developed in this MSc thesis is particularly targeted to assess the impact of software malfunction is cloud systems.

2.4 Tools

The development of a Software Fault Injection (SFI) is a process that can be accomplished through the use of tools for management and code analysis. In the following two sections, the available tools for both cases will be introduced.

2.4.1 Management of Software Code

In this section, a few tools to develop the SFI are presented and compared. The evaluated tools are Lex and Yacc, Eclipse CDT, GCC Parser and MCPP preprocessor. These tools promote a faster development of fault injector, and should perform some of the following tasks:

- Parse the software code;
- Construct the abstract syntax tree;

- Modify the code generated.

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 program. The Lex builds tokens and they are declared in the Yacc specification file. In order to use this tool, it would be necessary to define the tokens and grammar of the C language which would be laborious and time consuming.

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

- 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. Eclipse CDT allows the development of a fault injector in the JAVA language, which is more maintainable and easy to use, write, compile and debug.

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

MCPP is a portable C and C++ preprocessor with many features related with validation. Natella [Natella, 2011], used it as a workaround for some shortcomings of the GCC's C preprocessor. Currently, this project is outdated, once the last update was in May 2013.

Finally, Eclipse CDT Plugin was selected for this project, in standalone mode and only importing the necessary libraries. The decision to use Eclipse CDT Plugin was based on the maintainability of software developed in it, and the low learning curve that the developers need to modify it.

2.4.2 Analysis of Software Code

One of the tasks of any software fault injector is the selection of the code where the faults will be injected. This selection of the code or files where the software faults are injected is performed through the analyses of the code in the files. These analyses can be achieved based on the following metrics:

- **Number of Methods (NOM)** - Total number of methods defined in the selected scope;
- **Number of Fields** - Total number of fields defined in the selected scope;
- **McCabe Cyclomatic Complexity** [McCabe, 1976] - Counts the number of flows through a piece of code. Each time a branch occurs (if, for, while, do, case, catch and the ?: ternary operator, as well as the && and || conditional logic operators in expressions) this metric is incremented by one.
- **LSLOC** - Lines of code intended to measure statements, normally ended by a semicolon or a carriage return.

The metric selected for the fault injector proposed in this thesis was the McCabe Cyclomatic Complexity, due to measure the number of independent paths in code, which in practice means that can count the number of test conditions in a program, and at the same time because is simple to apply. To get the McCabe Cyclomatic Complexity, it was needed to perform a script to count the conditions, or use some tool that provides it. Bellow, it is addressed some tools that provide metrics to evaluate code, such as Metrics, CCCC, Code Analyser and Metriculator.

Metrics is an Eclipse plugin that provides metrics calculation and dependency analyzer. It measures various metrics with average and standard deviation and detects cycles in package and type dependencies and graphs them. Using it, it is possible to know the McCabe Cyclomatic complexity, but only works in Java and Apache2 is programmed in C.

C and C++ Code Counter is a tool which analyzes C++, Java and generates a report on various metrics of the code. Metrics supported include lines of code, McCabe's complexity and metrics proposed by Chidamber&Kemerer and Henry&Kafura.

Code Analyzer is a java application for C, C++, java, assembly, html, and user-defined software source metrics. It calculates metrics across multiple source trees as one project. It has a nice tree view of the project with flexible report capabilities, but, do not have McCabe Complexity metric.

Metriculator is another Eclipse plugin that creates static software metrics based on C++ source code. Some of metrics that are already implemented are McCabe Complexity, Logical Lines of Code and Number of Params, Efferent Coupling, Number of Members. This plugin has some advantages, such as:

- Smoothly integrated into Eclipse UI (Juno and earlier);
- Rich export functions (tag cloud image, HTML report, ASCII Text file, plain XML);
- Configurable limits per metric (e.g. maximum lines of code per function);
- Designed to easily add your own metrics (via separate plugin or via project contribution).

It was chosen metriculator, due to its calculate the metric previously selected, McCabe Cyclomatic Complexity. Moreover, the metriculator works with C++ language, working also with C language, since C is enclosed in the C++ language.

2.5 Hypervisor

In order to be able to setup a small scale cloud-based environment, it is necessary to refer to virtualization. Virtualization is a method of dividing the system resources provided by computers between different applications. A Hypervisor is necessary to manage the method of virtualization.

A Hypervisor [Fornaeus, 2010] or Virtual Machine Monitor is a program that creates and runs Virtual Machines. A Virtual Machine can be characterized as an efficient and isolated duplication of a real machine. Therefore, the Hypervisor is software that manages multiple Virtual Machines with multiple operating systems. The most important characteristics of a Hypervisor are:

- The management of one or more operating systems;
- Loading and booting these operating systems;
- Share memory;
- CPU power;
- The peripherals with each operating system;
- The isolation between Virtual Machines allows that in the event of a failure in one of the Virtual Machines, this will not bring down any other Virtual Machine.

As described in [Popek and Goldberg, 1974], Hypervisors can be classified as Native-Hypervisor or Hosted-Hypervisor.

The Native-Hypervisor, represented in Figure 3, has the following characteristics:

- Only provides the minimum set of functions;
- Runs directly on the host's hardware to control the hardware;
- Do not have operating system, only manage guest operating systems.

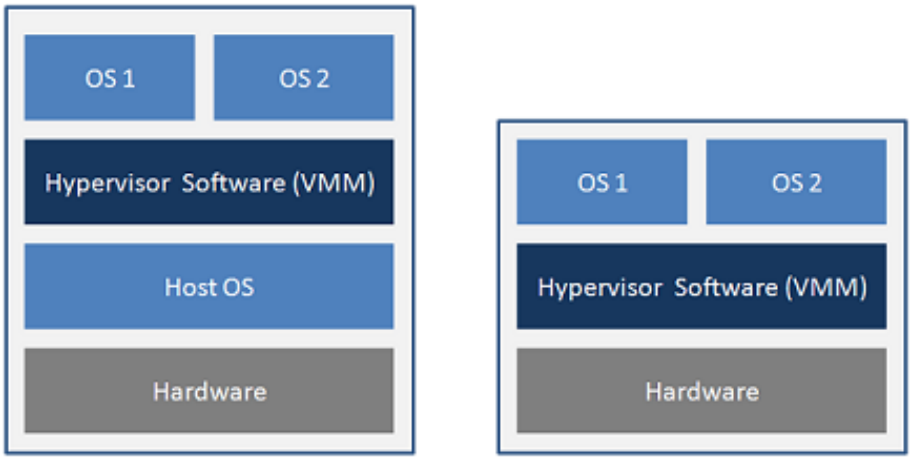

Figure 3: *Hypervisors: a Native-Hypervisor (on the right) and a Hosted-Hypervisor (on the left).*

Some examples of this type of Hypervisor are KVM, Hyper-V, VMWare ESX and Xen.

The Hosted-Hypervisor (Figure 3) runs above the operating system, as another computer programs. Some examples of this type of Hypervisor are VMWare Workstation [5], VirtualBox [6], Qemu [7].

The main difference between the Native-Hypervisor and Hosted-Hypervisor is the existence of an extra layer which is the operating system in the host. This extra layer can cause disruption in guests because the communication between devices is not straightforward.

2.6 WebServers

Cloud computing is used by public and private organizations. Most of its usage is based on webservers. There are multiple uses for webservers, such as simple websites or complex applications containing application programing interfaces (API). Due to the increasing usage of webservers, it is considered a good study case to assess the robustness of the Cloud. In this section, several webservers will be compared, and one webserver will be selected to perform tests.

Nowadays, all the websites of online services need an online server in order to: accept requests from clients, usually browsers; and provide answers, commonly HTML documents with information, images, etc.

In the segment of webservers, Apache WebServer leads the market with 36%, followed by Microsoft IIS with 27%, Nginx and Google's GWS with 17% and 2%, respectively, as can be seen on Table 2 [8]. It was detected a total of 901.002.770 websites and 5.579.077 web-facing computers.

Developer	November 2015	Percent	December 2015	Percent	Change
Apache	334,095,102	37.00%	320,676,759	35.59%	-1.41
Microsoft	244,906,586	27.12%	239,927,013	26.63%	-0.49
nginx	149,967,733	16.61%	157,001,018	17.43%	0.82
Google	19,622,624	2.17%	20,362,678	2.26%	0.09

Table 2: *WebServers Market Share of November 2015 and December 2015.*

From the webservers listed above, Apache WebServer and Nginx must be highlighted: Apache because of its utilization rate and Nginx due to its growth.

Apache WebServer, also named Apache 2 or HTTPD, was created by Rob McCool in 1995, when he was employee of NCSA (National Center of Supercomputing Applications). This is the most used webserver on the web since April 1996. Recently, it lost many users who have migrated to other webservers, although Apache continued to lead the segment of webservers. Please refer to Figure 4. The migration to other servers is related to the use of older versions, vulnerable and even unsupported, causing problems in the access of websites/services and in their management.

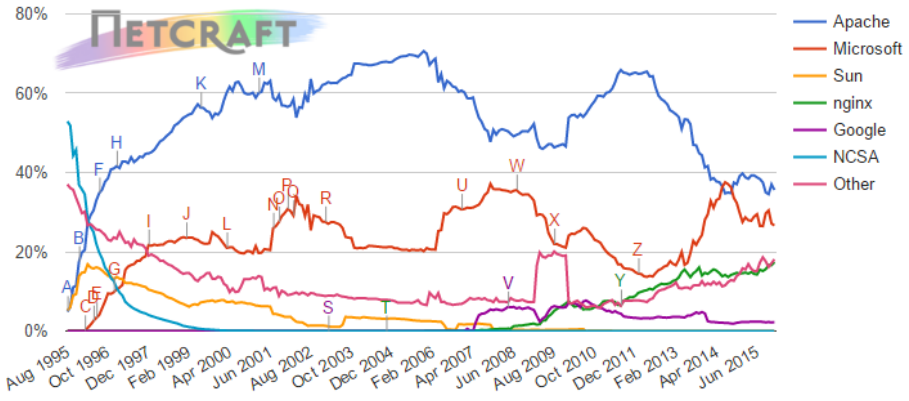

Figure 4: *Web server developers: Market share of all sites.[8]*

Nginx is one of the most powerful and stable webservers. Moreover, it is free, open-source and has high performance. This server has been designed and developed in 2005 by a Russian programmer Igor Sysoev. The main purpose of Nginx is stability, its easy configuration and low resource requirements at the level of hardware. The use of Nginx has been gradually increasing, according to NetCraft [8].

Considering that Apache is the most widely used webserver, failures will be injected in their source code in order to evaluate their behavior. In order to evaluate the effect of the failures injection, it is necessary to group similar results and assign them to a given classification, depending on a given scale, which is the subject discussed in the following section.

2.7 Analyze the effects

A group of injection of faults (in source code or not) will produce results. In this specific case, the injector BugTor was used to inject faults at source code level in order to obtain results. Afterwards, a careful evaluation of the results was mandatory. Other research papers have been selected and analyzed to choose the most important scales related with testing software in the Cloud. The scales that best framed to the evaluation of the results, although they have some gaps relating to their application directly in this work, are the CRASH Scale [Koopman et al., 1997] and the virtualized environments failure modes created [Cerveira et al., 2015].

Crash scale is a scale created by Koopman [Koopman et al., 1997] with the objective to group the faults of robustness tests according the severity of effect to the system user. This scale is centered on the user's vision of effect.

- **C**atastrophic - Operating System crashed or multiple tasks affected. This type of failure means that other tasks or the system crash or hang. It is usually solved by a system reset. They are detected using robustness benchmarks when the benchmark and starter tasks stay unresponsive to messages from the watchdog task;
- **R**estart - Single task or process hangs, requiring restart. It can be solved with a kill and restart of task, to put in the normal execution. It can be detected using robustness benchmarks when the task is still listed by the OS as an active task, but the benchmark task fails to respond to messages after a timeout interval;
- **A**bort - Single task or process aborts abnormally (i.e. "code dump" or "segmentation violation"). This type of failure is often caused by a memory access violation, when the task attempts to access a piece of memory (to write or read), that are reserved to another program or do not exist. If this happen during a system call, the exception is handled appropriately be the OS and is not considered an Abort failure because is not user-visible. It is detected by the robustness benchmarks when his tasks abort;
- **S**ilent - Test Process exits without an error code returned when one should exist. This type of failure happens when invalid parameters are sent thought an OS call and is not returned

any error. An example of this failure is when the call to open a file is done with the name NULL, can be done with success rather than return an error. It can be detected using the testing log and verify when the error codes have been generated and not returned;

- **H**indering - Test Process exits with an error code not relevant to the situation or incorrect error code returned. This type of failure can cause an erroneous recovery action. It is detected verifying when found an incorrect return code in the testing log;
- **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). Nevertheless, the severity is related to the type of system user and its point of view.

This *CRASH Scale* is a way to group the results of the effect of faults on an end-user system, mainly from the operating system perspective. In view of virtualized systems, there is the classification made by [Cerveira et al., 2015] that classify the failure modes represented from the external view (i.e., from the client side). This scale was created with the principle that the setup has two or more virtual machines running above the Hypervisor, and are injected some faults in one of the virtual machines, and observed the behavior of others. In their setup, [Cerveira et al., 2015] used a Web server named Apache2, to perform some requests involving computing a SHA1 hash. Obtaining the following levels:

- **Incorrect content** - It is characterized by the production of a syntactically correct HTML content with wrong values. This content can be seen by another machine or a human.
- **Corrupted output** - Described by a syntactically incorrect output, server fails to comply with the HTTP protocol, sending an invalid HTML code, or sends code which is not HTML at all.
- **Connection reset** - The communication between the servers and the client is reset.
- **Client-side timeout** - The client fails to receive any response within a defined period, causing a client-side timeout.
- **Hang** - The service stops producing responses to client requests. It can lead to all clients connected finish due to client-side timeout.
- **No effect** - It is not perceived any effect on the provided service by the user, both at level of performance as of correctness.

In conclusion, both scales are characterized by an end-user view and as he is aware of the errors that have happened. In the other hand, both can be distinguished based on the environment and setup of experiments. CRASH Scale [Koopman et al., 1997] is based in an Operating System and some tasks, and the [Cerveira et al., 2015] Classification is characterized by a system using virtualization, simulating the cloud and the communication between end-user and cloud.

3 Fault injector - BugTor

In this chapter is described all the work that have been done related with the development of the fault injector. Furthermore, a fault injector is characterized and described their operating principles, such as: faults to inject, way to inject, etc. Afterwards, the requirements for running the tool are presented and is explained how to use it. Finally, it is outlined how the fault injector was validated and its limitations.

In order to create a fault injector is necessary to define their characteristics, such as:

- Type of injector;
- Technique to use;
- Faults to inject;
- How to inject.

At the beginning of this project, it was considered to obtain information about the most representative faults in software. However, after discussion with supervisor, it was decided to take advantage of Durães' field study [Durães, 2005]. The use of this is an advantage due to this type of information rarely are available to the public and, this specifically have the main feature that is based in open source software developed in C. In that field study, can be seen information about the eighteen operators most representative in the open source software model. Nevertheless, after gathering the information and analysis, it was verified that not all the required information was available to implement the operators due to several reasons:

- Production of many mutations;
- Inconclusive definition of the operators;
- Little or missing information about the cases where they are applied;
- It can produce warnings or even errors while compiling;
- Low representation in relation to other operators (that can be seen in Durães' field study [Durães, 2005]).

In order to overtake some of these factors, it was necessary to get the data which Durães [Durães, 2005] have been used to perform his field study, or do a new one. However, due to time constraints, do a new one were unfeasible at this time. For these reasons, the five operators represented in the Table 3 were excluded and only the thirteen (listed in the Table 4), from the eighteen most representative, were implemented.

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 system. Whereas, in the spatial way, the fault is injected when it reaches the specified place where the particular operator can be applied.

This fault injector will be dubbed by BugTor due to the fact that it is a SFI that inject faults in software. In other words, the name for this injector came from "injector of bugs". BugTor emulated faults are similar to the real software faults made by real programmers who might lead to bugs. Therefore, this injector was developed in Java, using Eclipse CDT Plugin, and inject faults in C code using the SWIFI technique. The injection is based in its location, because only the code locations that validates all the constraints related to each one of operators, can be used to inject faults.

Operators	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.*

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 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 4: *Fault emulation operators.*

3.1 Operators

Concerning failures that will be injected, they are produced by applying operators, the Bug-Tor will inject thirteen of the most representative faults, previously specified by Durães & Madeira [Durães and Madeira, 2006]. According to them data-field results, to inject each one of the most representative faults is necessary to implement each one of the corresponding operators, as well as the matching constraints. These operators are specified individually below:

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 perform this removal, the constraints below need to be validated.

- **C01** - Return value of the function must **not** be used;
- **C02** - Call/Assignment/The if construct/The statements must **not be** the only statement in the block.

MIA - Missing if construct around statements

This operator simulates a missing *if* condition surrounding a set of statements. This causes the statements to be always executed and not just when the condition of *if* statement is true. The constraints to be applied in this operator are:

- **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.

MIEB - Missing if construct plus statements plus else before statements

This operator generates derivations of the source code of applications by removing *if* construct plus statements plus else before statements. In order to apply this operator, the following constraint must be verified first:

- **C13** - The if construct must **be** associated to an else construct.

This constraint does not exist in Durães & Madeira [Durães and Madeira, 2006] specification, but as this operator can not be applied in all situations, here it was specified and implemented. This operator can only be applied where the if construct has an associated else.

MIFS - Missing if construct and surrounded statements

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, it was needed to verify the following constraints:

- **C02** - Call/Assignment/The if construct/The statements 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.

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

This operator is based on the removal of part of a logical expression used in a branch condition. In order 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. In this case, was necessary to check the constraint **C12**.

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

This operator was omitted in the original specification, but as it can only be applied where the logical expression contains at least two sub-expressions, it was specified and implemented.

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

This operator emulates the removal of part of a logical expression used in a branch condition. In order 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 being evaluated as true. This operator has only one constraint:

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

The logical expressions can be located at *IfStatements*, *DoStatements* and *WhileStatements* and can have a lots of formats.

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 and for that, should be verified the following two constraints:

- **C02** - Call/Assignment/The if construct/The statements must **not be** the only statement in the block;
- **C10** - Statements are in the same block, **do not include** more than five statements, nor 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 is 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**.

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/Assignment/The if construct/The statements 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.

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/Assignment/The if construct/The statements 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.

MVIV - Missing variable initialization with a value
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/Assignment/The if construct/The statements 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.

WAEP - Wrong arithmetic expression in parameters of function call
This operator represents the modification of the expression used as parameter of a function call.

WPFV - Wrong variable used in parameter of function call
Operator WPFV 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.

WVAV - Wrong value assigned to a variable

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. In order to perform this, 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 are implemented according to their specification. To apply an operator, all the constraints related must be true. After apply each one of the operators in the code tree will be generated files with the modifications, named patches. These files are used in the testing process.

3.2 Constraints

In accordance with the Durães & Madeira [Durães and Madeira, 2006], the operators can only be applied when all the related constraints are true. In Figure 5, are represented all the existing constraints in the original specification. It is necessary to mention that these constraints have been specified using a generic programming paradigm in order to be used by more than one operator.

Constraints	Description
C01	Return value of the function must **not** be used
C02	Call/Assignment/The if construct/The statements 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, nor loops
C11	There must **be at least** two variables in this module

Table 5: *Fault emulation contraints defined by Durães [Durães and Madeira, 2006].*

In the Appendix C, some positive and negative cases of each one of the constraints can be seen. Moreover, it is possible to verify, after a brief analysis that the constraint **C07** is similar to constraint **C04**, one is the negation of another. This also happens with the constraints **C08** and **C13**. 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** have been implemented, it was needed to define the constraints **C13** and **C12**, because these operators do not have specified constraints and not be applied in all situations. The constraint **C13** was created because of the operator **MIEB** can not be applied to an *if* construct without an *else* construct. The constraint **C12** has been created too, because the operator **MLOC** can not be emulated in a branch with only one condition.

The absence of the constraints, represented in Figure 6, in the specification of Durães & Madeira [Durães and Madeira, 2006] may be related to the differences of G-SWFIT, the technique that Durães used, and the technique that was used in this fault injector, the SWIFI. The G-SWFIT is a technique which changes the binary code of the target to reproduce the sequence of instructions that corresponds to the required fault injected in the source level. And in BugTor the SWIFI are used directly at source code level.

Constraints	Description
C12	Must have **at least two** branch conditions
C13	The if construct must **be** associated to an else construct

Table 6: *Other constraints.*

3.3 WorkFlow and Implementation

In this section will be described the work that was done with the BugTor and the most important aspects of the implementation. One of the most important features of BugTor is the language in which it was developed, the Java language. This language provides some advantages, like as:

- Easy to learn;
- Object-oriented;
- Platform-independent;
- Distributed;
- Secure;
- Robust;
- Multithreaded.

After the study of some software that helps in the code editing (it can be seen at subsection 2.4.1), it is concluded that the usage of CDT was the best choice, given its potential. The CDT plugin provides some very useful characteristics and it was necessary to understand their workflow, their classes and their dependencies.

During this study, it was necessary to study the visitor pattern, which are typically named design-patterns. Visitor Pattern is a powerful behavioral pattern and it is used to manage algorithms, relationships and responsibilities between objects. From [Gamma et al., 1994], *"Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."*.

The CDT plugin makes use of Visitor Pattern to perform a pretty print of code tree, and the BugTor use the Visitor Pattern too, to perform the modifications with the operators. The use of this pattern:

- Allow the addition of new operations without the need of write the same switch statement over and over again;
- Provides a template for new operations to follow;
- Let the development of cleaner code, because the Visitor Pattern handle the conditional logic, instead of get the processing code and conditional logic in the same code.

After the initial analyses of the structure of CDT, it is necessary to begin the implementation of BugTor. To implement the BugTor, it is very important construct a good workflow. The main tasks performed by the SFI are:

- Read source code;
- Create Abstract syntax tree (AST);
- Verify all the constraints related to the current operator;
- Apply the operator in AST (only if all the constraints are valid);
- Create the patch with the modifications, comparing modified code with the initially source code.

Despite of the CDT Plugin have many attractive characteristics for this project, the implementation of the first operator with Eclipse CDT took a great effort. After some time spent to understand the tool and its class structure through Javadoc/Eclipse Documentation, it was even necessary to obtain more information from those who know and actually use it, by accessing the official mailing list: cdt-dev@eclipse.org. After exchanging some emails with Thomas Corbat, we conclude that Eclipse CDT does not allow the creation of a new code tree by changing the original tree. Moreover, to overcome this limitation, there are two possible ways:

- Use reflection to get the modifications from ASTWriter and pass it to ASTRewrite to get the code with the changes done;
- Get the source code of CDT and change it to avoid using reflection.

Reflection is a mechanism that permits the modification of structure and behavior dynamically. The systems are structured with two levels, the base level and the meta level. The meta level contains all the information related to the system properties. The base level has the information about application structure and its behavior. The modifications that have done at meta level are reflected to the base level structure and behavior. A good example of reflection pattern usage is the Constitution and the congress [Gamma et al., 1994]. The Constitution, matches to the meta level, have the information related to how the congress is to conduct itself. The congress, as base level, conducts itself according to the Constitution. The changes done in the Constitution change the behavior of congress. Initially, reflection was used to implement the first operator, despite of knowing that it is not a neat solution and is generally slower than equivalent native code. However, after understanding better the flow of Eclipse CDT, CDT's source code has been modified to avoid the use of reflection. However, during the implementation of the first operators, the tree is traversed using recursion, complicating the development of code and having the logic and the whole processing together. If the Visitor Pattern is used by the CDT to perform a pretty print of tree, why not take advantage of this pattern for the tree and the consequent injection of faults? Therefore, using the Visitor Pattern, makes the code simpler, cleaner and safer.

A diagram that represents an overview of the fault injection tool can be seen in the Figure 5. The fault injector starts by reading the source code, files typically coded in C or C++. The code is analyzed by the CDT and an AST tree is then created. In order to inject a fault, the injector finds the node where it can be injected (evaluating the truthfulness of all constraints of the operator concerned), and modifies it, according to operator specification. After that, the AST is rewritten, getting the code again, now with modifications. Finally, with the comparison of the two source codes (source code and code with mutations), it is made a *diff*, to obtain a summary of the changes made between files - patch. The decision to use the diff tool is related with the possibility of create smaller files with only the changes that are made, instead of using all the code with the modifications made in it.

3.4 Requirements

Due to the fault injector was deployed using Java and taking advantage of some features of CDT, a plugin for Eclipse. In order to use the BugTor, we must ensure some requirements. The main requirements are the installation of the following software:

- **Java 8** - Language used for the development of the BugTor, together with the plugin CDT.
- **Diff/Patch** - Tools used for create patches and manipulation thereof, based on the comparison of files and thus creating patches with differences (rows inserted, changed or removed).
- **Gcc** - Compiler for C language used to remove comments from the source code.

The use of injector in Linux is recommended instead of Windows, since it is already included in the system path some tools necessary for their correct usage, such as diff, patch and gcc.

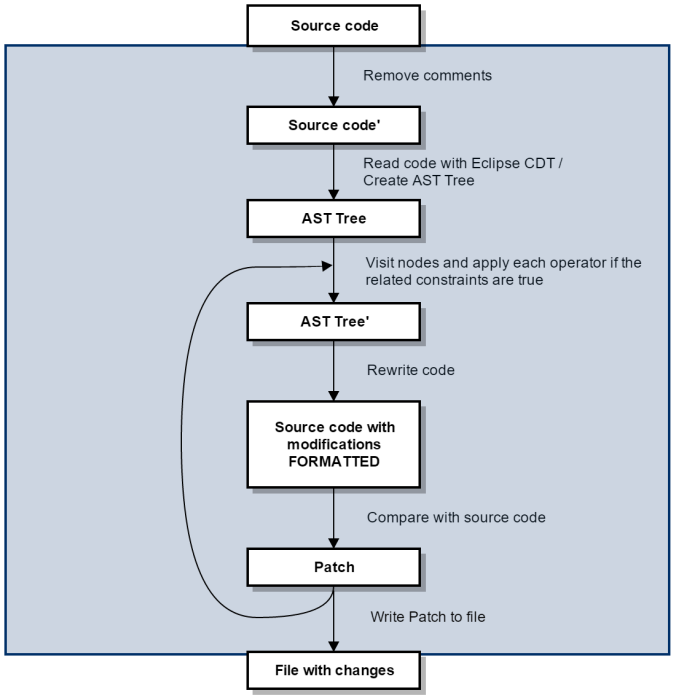

Figure 5: *Workflow of the injection tool.*

3.5 Usage

In order to be able to use BugTor, it is noteworthy some particularities related to its usage. For example, the structure of command to run it:

SYNOPSIS: java -jar FaultInjetor.jar FILE [OPTION...]
FILE .c .cpp
OPTIONS

- **-s Silent option** - Not print the readable code by CDT. This is related to the limitations of fault injector, presented in Section 3.7.
- **-sp Silent patch option** - Not print information related to the created patch, number, modifications, etc.
- **-d Debug option** - Print code before the application of each operator, as well as the code to change or remove, even as the final code, after the modifications performed by the operator.
- **-o=OPERATOR** - Apply the BugTor only to an operator. The operators can be seen in the Table 4. OPERATOR is case-insensitive.

Below are presented some examples of execution of fault injector, for different situations:

- Execution of BugTor only with the application of operator MIFS, in file “file.c”:

```
1 $ java -jar BugTor.jar file.c -o=MIFS
```

- This execution differs from the previous one by the activation of DEBUG option. With DEBUG option activated, showing information related to the steps that are doing by the fault injector:

```
1 $ java -jar BugTor.jar file.c -d -o=MLAC
```

For a correct usage of this software, it is necessary having some knowledge regarding the execution of commands through the console.

3.6 Verification and Validation

Verification and Validation (V&V) is an important process in the development of software, and this implies the need to be done by external people. In this case and because the software developed are done during a dissertation, this phase must to be done by the student which do the development.

During the development of this fault injector, a script to perform automated tests to many source codes was done. This script was created in Python language and it runs the BugTor, generating derivations to many files with code through application of each one of the operators. The main function of this script is count the number of patches generated with each one of the operators. Using this, it can be sure that when a new operator is developed, the other operators not be affected (the number of patches counted can not change from the last iteration). This technique is called regression testing [Wong et al., 1997], since when a new operator or constraint is developed is verified with that script if the previous implementation of operators and constraints is not affected.

Moreover, after the implementation of each operator, during the development of BugTor, were checked the number of patches created, as well as its content. For example, in the case of operators MLAC and MLOC, it was necessary to verify the branches containing the operators && and ||, and the removal of each one of this elements of branch will produce a new patch.

The files for testing are selected based in several characteristics, such as:

- Function calls and number of parameters;
- Conditionals, only *if*, *if* with *else*, etc;
- Number of statements within a function, or in a block;
- Number of logical expressions used in a branch condition and different operators.

The number of files for the testing process was increased during the development process because when it was found a new case that was not covered by the current code files, it was added to the test. At the beginning were approximately twenty files, and with the progress of the development process, quickly reached more than one hundred.

3.7 Limitations

In accordance with another projects, during the development of this injector some problems have happened. Some of them derived from choices taken. However, most of them were overcome using the best programming practices.

The executed test that have been done during the development showed some gaps, more specifically related with the manipulation of code that depends from macros, if the specific macro was defined or not. The application of this injector in real complex code that includes macros stated that the CDT not represent the macros in the created tree, not being possible to make changes directly, using the features of CDT.

This limitation can be overcome through two ways, such as:

- Solve the macros using per example *gcc -e file*. The macros will be solved and the output can be the input of BugTor. It can not work to some application which evolve a very big number of dependencies.
- Reduce the amount of code to be tested, selecting only a function, or more parts that are not affected by macros.

If the injector is used in code with problematic macros, it warms the user, showing the message “Verify limitations with macros!”. On the other hand, if the code where the BugTor will be applied do not have macros, or only have macros at the beginning of the code, like the code represented in Listing 28 the BugTor create the patches without any problem.

Listing 29 (Appendix D) shows some code of the module *mod_rewrite*. This code has the macros *#ifdef*, *#else* and *#endif*. If the macro is already defined, the CDT will read the code between the *#ifdef* and *#else*, otherwise, the CDT will read the code between the *#else* and *#endif*.

To reduce the impact of macros in fault injector, a workaround has been created to correct most of the problems, but empty patches can be created due to the replacement of macros in the original files, in rare cases. This workaround is based in the comparison of the original code file with the code file changed, and it resets the macros and the related code where it was removed.

4 Work and implications

When this project has been started, it was necessary to choose the technique that is used to inject faults. However, as stated earlier, at 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 [Cerveira, 2015], the project was modified to inject software faults. After taking that decision, it was chosen the technique that we could 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. There are some available tools, which inject faults in execution environment (e.g. by [Durães and Madeira, 2006]). Also, Natella [Natella, 2011] had previously coded a tool which injects faults at source code level, using MCPP as a C preprocessor during his PhD thesis.

Taking all these factors into account could lead to the conclusion that choosing to inject faults at object code level (i.e., during compilation time) would be the best option, in what concerns research innovation. However, the usage of this technique in the development of this fault injector and the evaluation of the robustness of the cloud would be a great effort 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 of the existence of Natella's tool [Natella, 2011] to inject faults at source code, the injector under development use the capabilities of Java Language, such as the maintainability and its easy of use, to inject faults in source code coded in the C and C++ language.

With BugTor, the faults will be injected in C and C++ code because of the extensive knowledge of the supervisors with this programming language and it was developed because of the work already done by Durães & Madeira [Durães and Madeira, 2006]. The specification of the operators and related constraints was based on a field study of open source software coded in that languages. After that, the software possibilities to parse the code and get the AST are evaluated. CDT Plugin has been chosen due to its features, such as source navigation, source code refactoring, code generation and abilities of the student in programming in Java Language.

To do this project is necessary to develop three separated modules:

- Generate the derivations of main code of selected programs (function performed by BugTor, Chapter 3);
- Compile the programs with injected faults, by using make file;
- Verify and analyze the effect of produced faults.

These modules are enough to perform tests in a Cloud Computing environment.

In Figure 6, it can be seen an overview of the main decisions taken during the first semester, and the development of injector, including its workflow and implementation. The decisions, that have been taken related to the development of injector (derivations of the source code), are already described at Chapter 3, more specifically at Section 3.3. To perform experiments, Apache was selected, since it is the most widely used webserver, can be seen more information about Apache at Section 2.6.

After get the derivations of the source code, it was needed to apply them at the application source code, compile and install it.

Normally, the applications have the code divided into files in order to increase the modularity and simplify the structure thereof. Furthermore, it is necessary to view the effect of fault injection from the final user view, and the injection in random files, or in all the source code files make this type of experiments unfeasible, since they can take a long time, or even not cause any

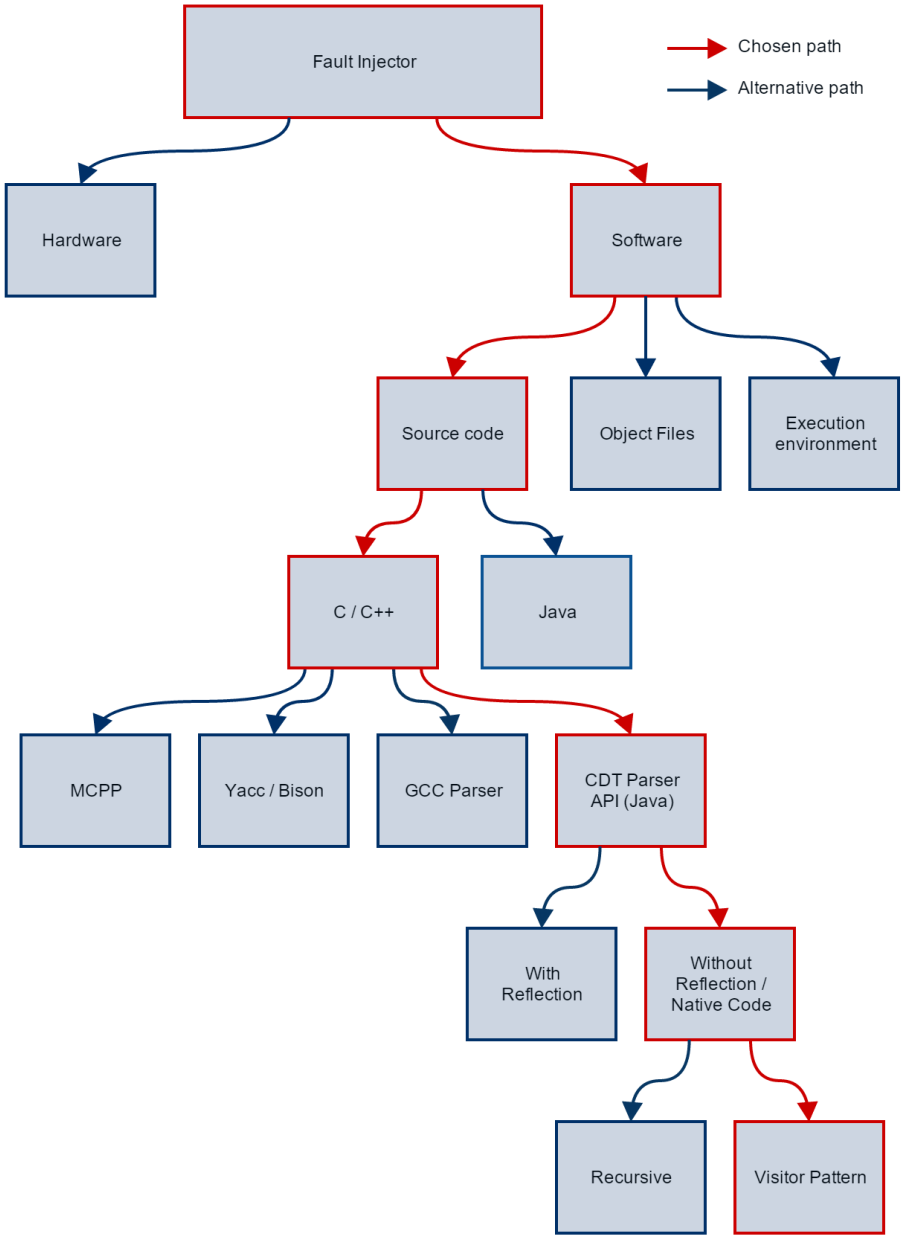

Figure 6: *Decision tree of first semester.*

kind of anomalous behavior. For this reason, it was necessary to define the file(s), or the module(s) to be evaluated through fault injection. There are several ways to make the file selection to inject faults, such as the code metrics (described at Section 2.4.2), the most modified code in the repository (usually named churn), the cohesion and coupling. To accomplish this task was selected the McCable Cyclomatic Complexity, since this code metric measures the number of independent patch in a program. Some advantages of this metric are: measure the minimum effort and best areas of concentration for testing; guide the testing process by limiting the program logic during development; and are easy to apply. In the Table 7 can be seen the files with most McCabe Cyclomatic Complexity in source of Apache. The other code metrics can also be seen, for example LSLOC (Lines of code intended to measure statements, normally ended by a semicolon or a carriage return) or Number of efferent coupling per type. All the files that begins with "mod_" are modules of Apache and they need to be activated to Apache use them.

After the measure of the McCabe Complexity, it is necessary to setup the system, and configure the Apache WebServer to use the module *mod_rewrite.c* (the module of the file with greater

Scope	Coupling	NbParams	NbMembers	McCabe	LSLOC
mod_rewrite.c	41	154	67	937	2269
core.c	26	326	5	653	2216
mod_include.c	37	135	55	641	1868
util.c	2	252	2	607	1565
mod_dav.c	3	136	10	603	1861
proxy_util.c	4	198	13	531	1554
mod_negotiation.c	7	99	44	481	1178
mod_proxy.c	0	165	0	453	1400
mod_autoindex.c	10	96	41	423	1133
util_ldap.c	0	148	0	408	1333

Table 7: *Apache2 source analisys - sorted by McCabe Cyclomatic Complexity.*

McCabe Complexity, that curiously is also which has a larger number of lines and a larger coupling). The module *mod_rewrite* is a flexible and powerful module of Apache HTTP Server, that permits to perform the mapping of arbitrary urls to internal structured urls, based in an unlimited number of rules. The rules are created using regular expressions. Some examples of their usage in this module are:

- Mapping a url to a filesystem path;
- Redirect one url to another url;
- Invoke an internal proxy fetch;
- Mapping the url parameters to a url with class/method/param/param;
- Redirect errors to a file (e.g. in case of a not found file or directory).

The rewriting can be done based on server variables, environment variables, HTTP headers, or time stamps.

In order to carry out as soon as possible the experiences, it was decided to use a Hosted-Hypervisor, instead of a native Hypervisor, thus speeding the experiments. The Hypervisor selected was the VirtualBox due to its easy to use and by student's knowledge.

The next step was the generation of derivations of mod_rewrite.c code, but some troubles appeared, related to the content of file mod_rewrite.c - macros. It is the current limitation of BugTor (more information and description regarding the solution found in Section 3.7).

The choices done regarding with the testing environment can be seen in the Figure 7.

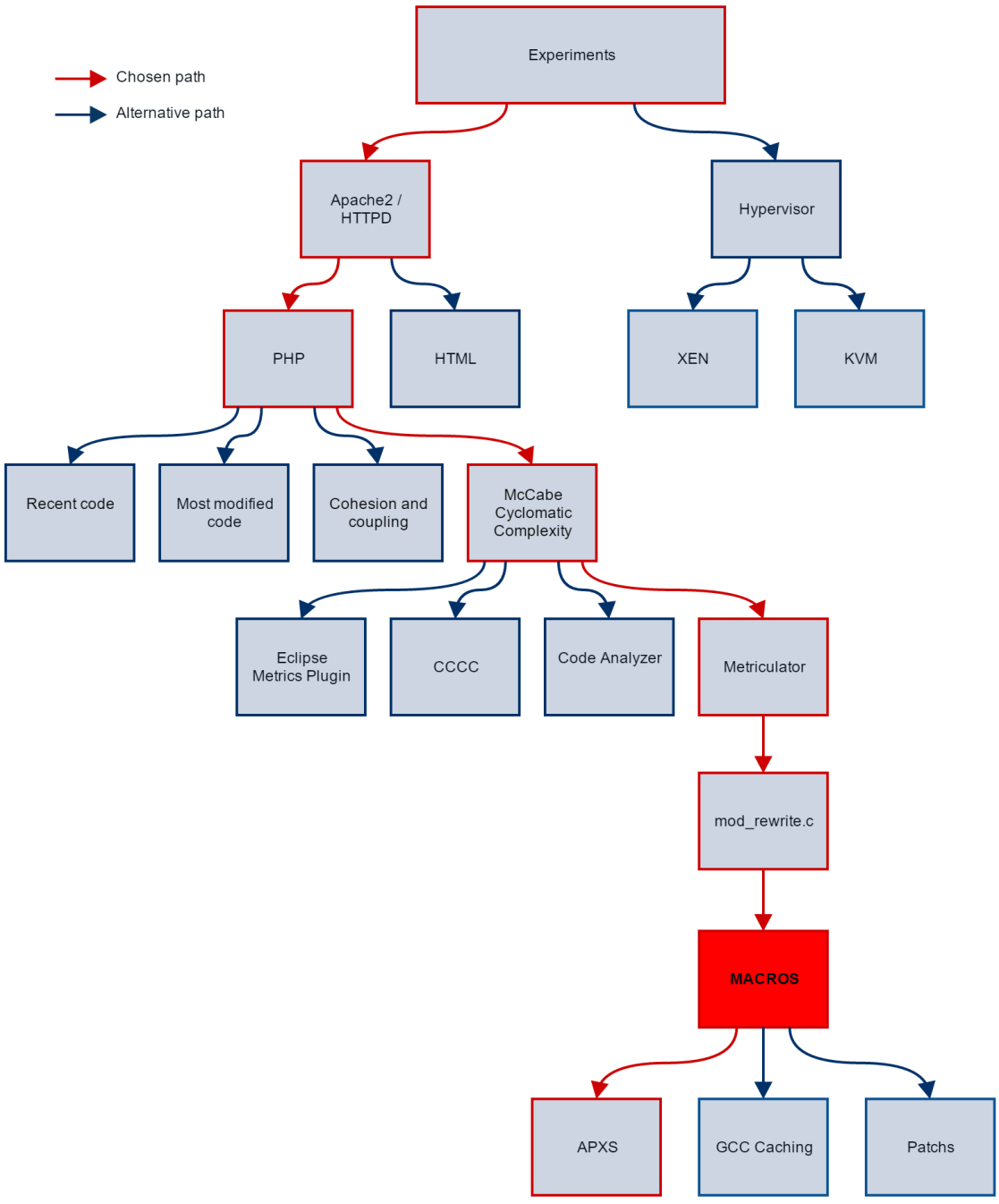

Figure 7: *Decision tree of second semester.*

5 Experimental Results

According to the original plan, in order to assess the robustness of the Cloud, it was needed to create a physical environment or emulate one. In this dissertation, the experiments were focused on fault injection in Apache Web Server and collecting information about its behavior. In Figure 8, it is represented the two environments that have been done during this project. The first scenario is in normal condition without any kind of failure. The second is with faults introduced by BugTor in the Apache. Depending on the type of fault injected into the Apache, it will have different behaviors, which will be evaluated and classified.

These two scenarios were chosen taking into consideration the following:

- Emulate a Cloud environment without and with the presence of faults;
- Perform some tests in the two scenarios and compare them to get conclusions;
- Use hypervisor and virtual machines as the real Cloud.

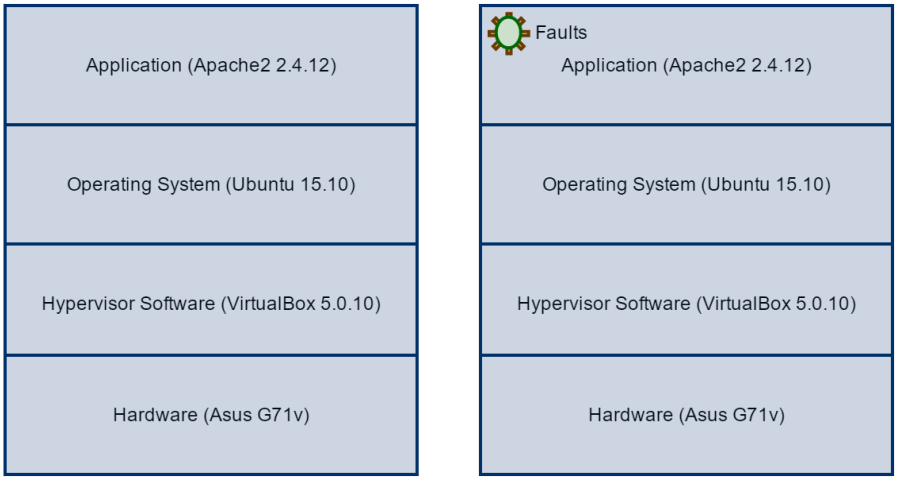

Figure 8: *First scenario (in the right) and second scenario (in the left).*

The following tests have been performed in the two scenarios:

- Checking if Apache is working correctly by making requests to the main page of Apache (after installing Apache the first time, is the *index.html*);
- Verifying if the PHP is working with requests to the page *phpinfo.php* (this page has several information of the running setup);
- Send four parameters in the url and print them using php;
- Perform some benchmarks using php performance benchmarks [9] (tests with strings and arrays);
- Obtain some system information to verify if the system is running correctly.

These tests, represented in the script in the Appendix E, are the most basic test that can be done in order to check the operation of Apache, in various situations:

- Requests to files in the *DocumentRoot* of Apache, HTML and PHP;
- Requests with some parameters in order to effectively test the behavior of module mod_rewrite. In this case are selected four parameters to send in the request, and they will be show at the content of PHP page returned. This is the more important test due to the effective use of mod_rewrite. ;
- Some little benchmarks to PHP functions.

It was scheduled to run more tests, particularly with JMeter in a more complex scenario, with two virtual machines. Due to time constrains, it was not possible the completion of the new scenario and therefore, it was not possible to make the associated tests.

5.1 Setup

The above mentioned experiments were performed in the student's computer, Asus G71v, which has the specifications referred in the Table 8 below. It should be noted that the experiments were performed at the same time as daily tasks, such as checking email, browsing, writing, programming, etc.

Below, it is possible to check the virtual machine specifications too - Table 9.

CPU	Intel Core 2 Duo T9400 2.53Ghz
Ram	8GB DDR3 1066Mhz
SSD	OCZ Vertex2 128GB
HDD	Seagate Momentus XT 750GB
Virtualization Technologies	VT-x, Nested Paging

Table 8: *Hardware specifications.*

CPU	2 Cores
Ram	2048MB
HDD	20GB
Operating System	Ubuntu 15.10

Table 9: *Virtual machine specifications.*

The main idea of the setup configuration is to have the Apache Webserver installed and allow the use of pages in HTML and PHP. In order to make Apache work properly, the following is necessary:

- Install Apache2 HTTP Server (version 2.4.12);
- Install Apache Portable Runtime;
- Install Perl Compatible Regular Expressions (PCRE);
- Install PHP: Hypertext Preprocessor (PHP5 - including the package libapache2-mod-php5);
- Some configurations at Apache (p.e. enable mod_rewrite module).

At the beginning of setup configuration, it was a need to download the sources of software, compile and install it. However, after some experiments, the script of compilation, installation and configuration takes more than ten minutes to perform one experiment, which is time-consuming and unaffordable. Therefore, it was found a solution to reduce the execution time

of each experiment, a kind of mechanism which is able to bypass the runtime. After some conversations with supervisor, the solution found was to use mechanisms of caching in compiling the Apache source, because we need to compile Apache each time that the mod_rewrite.c file is changed, taking over more that 90% of the runtime. After some time spent in the study of the GCC caching mechanisms, and attempt to apply it in Apache, it was concluded that given the complexity and high number of dependencies of module mod_rewrite, its use would not be feasible. Following this and given the unsolved problem, after further research, we found a APache eXtenSion tool (APXS). This tool makes possible the installation of new modules without the need to compile all the Apache again, and that actually what was intended. The use of APXS made possible the execution of experiences in useful time.

In order to setup properly Apache WebServer and consequently the rewriting module, it was necessary to make changes at the following files: *000-default.conf*, *apache2.conf*, *httpd.conf*, *.htaccess*. The rewriting module was based in rules and that rules for that are in file *.htaccess*, as represented in Listing 5.1.

```
1  # .htacess mod_rewrite
2
3  RewriteEngine On
4
5  # if directory exits, not apply any rule
6  RewriteCond %{REQUEST_FILENAME} !-d
7  # if file exits, not apply any rule
8  RewriteCond %{REQUEST_FILENAME} !-f
9
10 RewriteRule ^(.*)$ index.php?url=$1 [QSA,L]
```

Listing 5.1: *Configuration of mod_rewrite rules in .htacess file.*

The first command line *RewriteEngine On* activates RewriteEngine, if it is not already activated. The next two lines "RewriteCond %REQUEST_FILENAME !-d" and "RewriteCond %REQUEST_FILENAME !-f" specify that just if the file with the specified name in the browser does not exist, or the directory in the browser does not exist, the rewriting proceed to the rewrite rule below. The last and most important rule are "RewriteRule ^(.*)$ index.php?url=$1 [QSA,L]" because if the file or directory does not exist, the Rewrite Engine will change the request based in the rule. The *QSA*, in the rule, means *Append query string*. In other words, each parameter in the url will be passed to the destination url as parameter, first parameter as $1, second parameter as $2, third parameter as $3 and so on. The *L* means that is the *Last* and RewriteEngine can stop processing rules, there are no more rules to process.

5.2 Classification of failures

In the beginning of this work, a study was done on two severity scales of the behavior of two different systems: a computer system and a cloud scenario. Both scales are not applicable to evaluate the results of the setup in question, because the components of the system are different. In the first system only exists a computer with some processes or tasks, and in the second, a cloud environment with a native-hypervisor and are injected faults in hypervisor. In this case, we have a computer with the emulation of a Cloud service and do the injection of software faults in a Web Server, named Apache. After the injection of several faults, the behaviors were evaluated using three steps. In the first are selected only the unique behaviors, in second step are analyzed and compared to create a classification, and in the end, a script will execute to evaluate all the behaviors, one by one, through the classification. The behaviors were being grouped in three categories:

- **C**orrect - The webserver;
- **W**rong output - Occurs when the webserver returns incorrect information, or no information;
- **A**pache error - All the errors directly related with the behavior of apache2, not found with/without correct url, Internal Server Error, Ok, Forbiden, Bad request.

After the classification, all the results need to be evaluated to can take conclusions about them. In the Appendix F is showed some behaviors of Apache and their classification.

5.3 Results and Analysis

During the fault injection stage, the BugTor evaluates the source code nodes to identify where can apply each one of the operators (all the constraints, related with the operator, need to be true in that location). For that reason, by evaluating the code of the Apache module "mod_rewrite", 1474 locations were identified to inject faults.

In the Table 10, it can be seen the number of patches created by each one of the operators. It should be noted that the number of patches created by the application of operator MLPA is quite large, as can be seen in Figure 9, due to it removes a small part of algorithm. That part of algorithm is any combination of up to five function calls and/or statements. It produced more than 1/3 of patches created.

Operator	Nr. of Patches
MIFS	239
MLAC	79
MFC	162
MIA	260
MLOC	41
MLPA	526
MVAE	25
MVAV	15
MIEB	54
MVIV	14
WVAV	24
WAEP	34
WPFV	1
Total	**1474**

Table 10: *Number of patchs.*

	Number	Percentage
No effect	1261	85,55%
With effect	213	14,45%
Experiments	**1474**	**100%**

Table 11: *Results of experiments.*

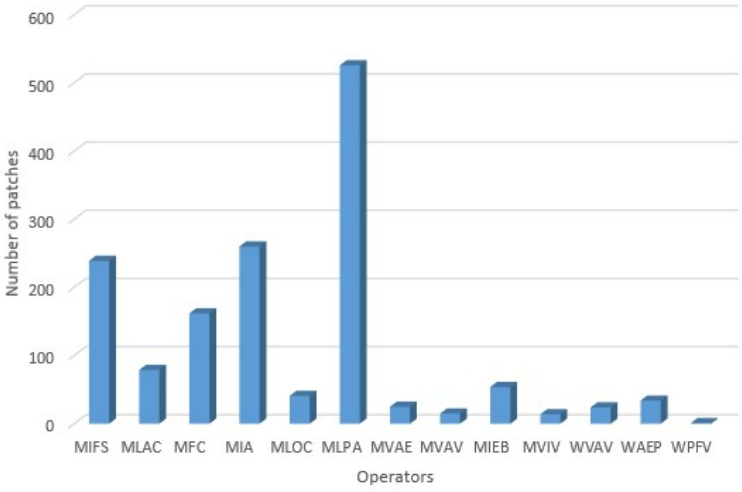

Figure 9: *Number of patches by operator.*

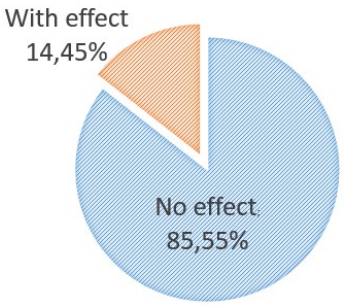

Figure 10: *Results of experiments.*

On the other hand, it is curious the existence of only one patch created through the application of operator WPFV. This operator is based on the replacement of a function parameter by another of the same type. This limitation is related to the environment in which the fault is injected, since the lack of it could trigger compilation errors, at the source code environment. The existence of only one patch is related to the need of parameter changed to be exactly of the same type of initial parameter. In C and C++ languages, the programmers create their own structures of a specific type, to be easier to represent the data in the application. For this reason, the types of variables used are quite different in a function and its number of occurrences low.

The injected faults produced anomalous effects in 213 trials (as can be seen in the Table 11 and Figure 10), corresponding to 14,45% of injected faults. The number of experiences is justified due to the constraints that need to be evaluated in each potential local in the code. If all the constraints of one operator are valid, then the operator will be applied.

After an initial assessment of the results of experiments performed, it was needed to verify each one of the anomalous behavior and identifying correctly what differences exist between each one and its correct behavior. The anomalous behaviors of the performed tests are showed in the Table 12.

#	Apache	PHPInfo	Out	PHPBench	System	N
1	W	W	W	W	C	86
2	C	C	A	C	C	61
3	A	A	A	A	C	25
4	C	C	W	C	C	19
5	W	W	C	W	C	19
6	C	W	A	W	C	1
7	A	W	W	W	C	1
8	A	A	A	C	C	1

Table 12: *Results of experiments by behavior (Ordered by number of occurrences).*

Classification of failures	
A	Apache Error
C	Correct
W	Wrong output

Table 13: *Classification of failures.*

They are ordered by the number of occurrences - **N**, divided into the following categories:

- **Apache** - This is the most basic test that should be done at Apache after installation. It is based on a request to *index.html*. The response is typically a page with some information about the Apache installation, with the string: “It works”, at the beginning;
- **PHPInfo** - This test was done to check the operation of PHP. It gives a large amount of information to the user, related to the configuration settings, PHP version, OS version information, etc;
- **Out** - Request with parameters that will be shown at the response;
- **PHPBench** - Verify and measure the time that certain PHP functions take related with the manipulation of strings and arrays;
- **System** - Tests at the system to verify if it works properly.

It can be seen that the operation of Apache is affected by the faults introduced too. However, there is no rule that rewrite files when they are available in the directory, which is the case of Apache “It works!”.

In the Tables 15, 16, 17, 18 are shown the individual results for each one of the performed tests. In these tables, the behavior was not restricted to the initial classification due to the very different types of errors in the Apache Web Server, see Table 14.

#	Description
1	Bad request
2	Empty
3	Forbidden
4	Found
5	Internal Server Error
6	Not found - url OK
7	Not found - wrong url
8	Apache error - Ok
9	Wrong output

Table 14: *Kind of behaviors. Examples in the Appendix F*

	Behaviors									
	1	2	3	4	5	6	7	8	9	**Total**
MIFS	0	4	0	0	2	0	0	0	2	8
MLAC	0	0	0	0	0	0	0	0	1	1
MFC	0	2	0	0	1	0	0	0	0	3
MIA	0	3	1	0	11	0	0	0	3	18
MLOC	0	0	0	0	1	0	0	0	0	1
MLPA	0	68	1	0	6	0	1	0	14	90
MVAE	0	4	0	0	1	0	0	0	2	7
MVAV	0	0	0	0	0	0	0	0	0	0
MIEB	0	1	0	0	1	0	0	0	1	3
MVIV	0	0	0	0	1	0	0	0	0	1
WVAV	0	0	0	0	0	0	0	0	0	0
WAEP	0	0	0	0	0	0	0	0	0	0
WPFV	0	0	0	0	0	0	0	0	0	0
Total	0	82	2	0	24	0	1	0	23	132

Table 15: *Apache tests.*

In the Table 15, it is possible to see that the Apache test have 132 anomalous behaviors from the 213 trials with effects. The operator that causes most failures is the MLPA, being related to the number of patches created, already explained. The most frequent behavior is a *Empty* response The other most frequent behaviors are the *Internal Server Error* and *Wrong output.* There are two other anomalous behaviors but with a very low frequency, a *Forbidden* and a *Not found - wrong url*, with frequency two and one, respectively.

The results obtained in PHPInfo and PHPBench tests, represented in the Tables 16 and 17, were similar. This happens because the PHPInfo and the PHPBench are based in two pages with different PHP code and the rewriting process for both are the same. This is related with the incidence of these tests on the same principles, the same workflow in Apache.

	Behaviors									
	1	2	3	4	5	6	7	8	9	**Total**
MIFS	0	4	0	0	2	0	0	0	2	8
MLAC	0	0	0	0	0	0	0	0	1	1
MFC	0	2	0	0	1	0	0	0	0	3
MIA	0	3	1	0	11	0	0	0	4	19
MLOC	0	0	0	0	1	0	0	0	0	1
MLPA	0	69	0	0	6	0	1	0	14	90
MVAE	0	4	0	0	1	0	0	0	2	7
MVAV	0	0	0	0	0	0	0	0	0	0
MIEB	0	1	0	0	1	0	0	0	1	3
MVIV	0	0	0	0	1	0	0	0	0	1
WVAV	0	0	0	0	0	0	0	0	0	0
WAEP	0	0	0	0	0	0	0	0	0	0
WPFV	0	0	0	0	0	0	0	0	0	0
Total	0	83	1	0	24	0	1	0	24	133

Table 16: *PHPInfo tests.*

	Behaviors									
	1	2	3	4	5	6	7	8	9	**Total**
MIFS	0	4	0	0	2	0	0	0	2	8
MLAC	0	0	0	0	0	0	0	0	1	1
MFC	0	2	0	0	1	0	0	0	0	3
MIA	0	3	1	0	11	0	0	0	4	19
MLOC	0	0	0	0	1	0	0	0	0	1
MLPA	0	69	0	0	6	0	0	0	14	89
MVAE	0	4	0	0	1	0	0	0	2	7
MVAV	0	0	0	0	0	0	0	0	0	0
MIEB	0	1	0	0	1	0	0	0	1	3
MVIV	0	0	0	0	1	0	0	0	0	1
WVAV	0	0	0	0	0	0	0	0	0	0
WAEP	0	0	0	0	0	0	0	0	0	0
WPFV	0	0	0	0	0	0	0	0	0	0
Total	0	83	1	0	24	0	0	0	24	132

Table 17: *PHPBench tests.*

	Behaviors									
	1	2	3	4	5	6	7	8	9	**Total**
MIFS	1	5	0	0	4	3	0	0	5	18
MLAC	0	0	0	0	1	2	0	0	0	3
MFC	0	3	0	0	1	2	0	0	1	7
MIA	1	4	1	1	13	17	0	2	3	42
MLOC	0	0	0	0	1	0	0	0	1	2
MLPA	0	73	0	0	8	23	1	0	2	107
MVAE	0	5	0	0	1	0	0	0	1	7
MVAV	0	1	0	0	0	0	0	0	0	1
MIEB	0	1	0	0	1	2	0	0	1	5
MVIV	0	0	0	0	1	1	0	0	0	2
WVAV	0	0	0	0	0	0	0	0	0	0
WAEP	0	0	0	0	0	0	0	0	0	0
WPFV	0	0	0	0	0	0	0	0	0	0
Total	2	92	1	1	31	50	1	2	14	194

Table 18: *Out tests.*

In the Table 18, it is possible to see the results of tests with some parameters. It is also possible to verify that four anomalous behaviors just happen in this test, *Not found - url ok*, *Bad Request*, *OK* and *Found*. As might be expected, this is the test where there are more errors due to greater use of module "mod_rewrite". All the requests are rewritten, and the parameters are presented on a page in PHP.

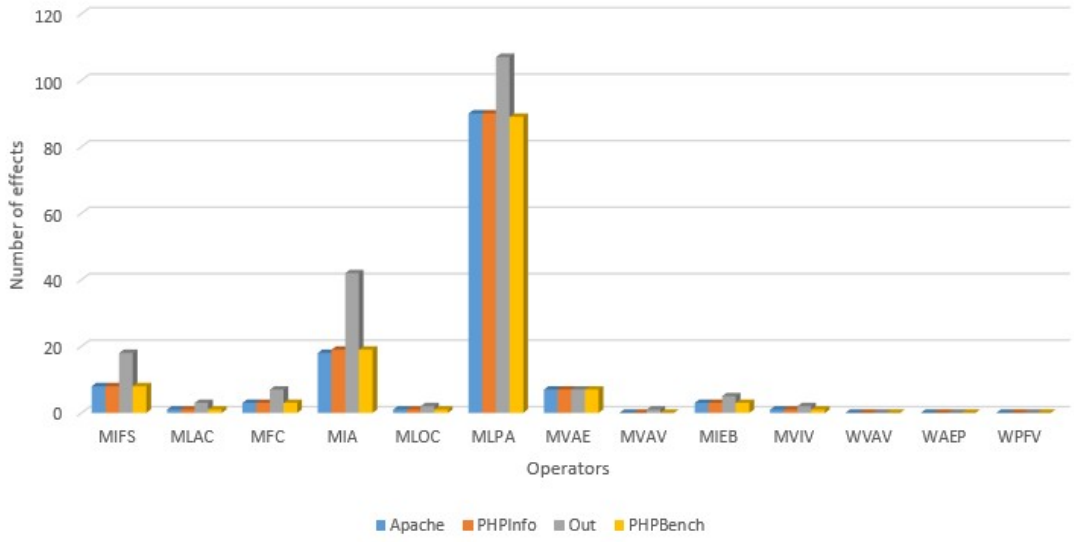

Figure 11: *Effects by patches.*

In the chart, represented in the Figure 11, it can be verified that the operators that produced more errors were MLPA, MIA, MIFS and MVAE. The first three correspond also to those who injected more failures.

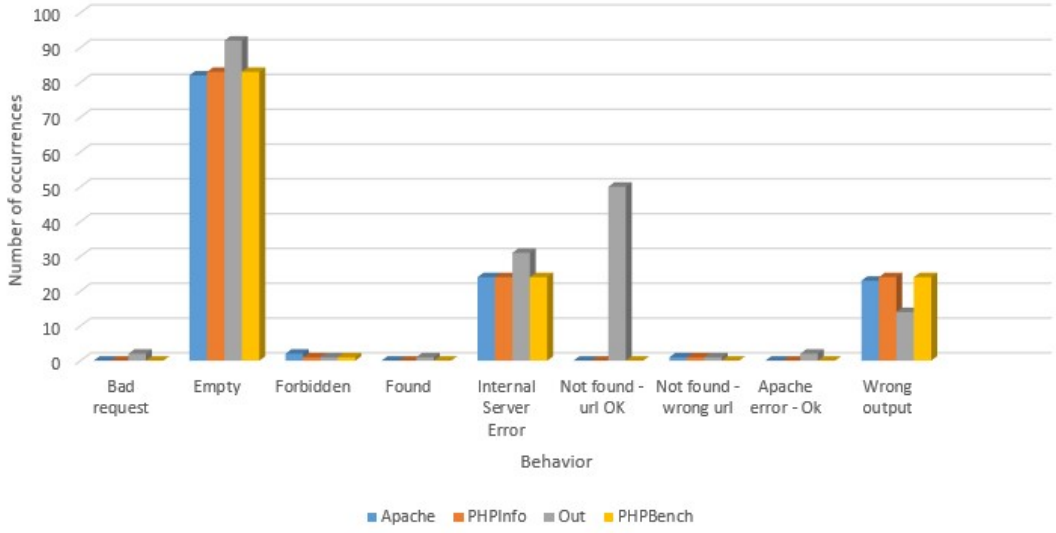

Figure 12: *Effects by behavior.*

As discussed above, the most anomalous behaviors occurred during the test Out, which is represented in Figure 12. In the same figure can be verified that the anomalous behavior that happens more often is the behavior *Empty*.

Moreover, it should be noted that the error *Not found - url ok* occurs often. This behavior means that the request is done correctly, and received properly by the Apache, but for some reason it is not rewritten correctly, and because there is no file by that name and in that directory, returns *Not Found*.

#	Apache	PHPInfo	Out	PHPBench	N
1	Empty	Empty	Empty	Empty	82
2	Correct	Correct	Not found - url OK	Correct	49
3	Internal Server Error	Internal Server Error	Internal Server Error	Internal Server Error	24
4	Wrong output	Wrong output	Correct	Wrong output	19
5	Correct	Correct	Wrong output	Correct	10
6	Correct	Correct	Empty	Correct	9
7	Correct	Correct	Internal Server Error	Correct	7
8	Wrong output	Wrong output	Wrong output	Wrong output	4
9	Correct	Correct	Ok	Correct	2
10	Correct	Correct	Bad request	Correct	2
11	Not found - wrong url	Not found - wrong url	Not found - wrong url	Correct	1
12	Correct	Wrong output	Not found - url OK	Wrong output	1
13	Forbidden	Empty	Empty	Empty	1
14	Correct	Correct	Found	Correct	1
15	Forbidden	Forbidden	Forbidden	Forbidden	1

Table 19: *Results of experiments by specific behavior (Table 14, Ordered by number of occurrences).*

In the Table 19, it is possible to verify that the results are the same in the four tests, involving 111 (sum of number of occurrences of lines 1, 3, 8 and 9) of 213 cases. Moreover, this only happens to the errors: *Empty*, *Internal Server Error*, *Wrong output* and *Forbidden*. In the same table, it is possible to see that only the test Out is erroneous in 81 experiments (visible in lines 2, 5, 6, 7, 9, 10, 12 and 14), the others are correct. The produced errors have many types, but the errors: *Not found - url ok*, *Bad Request*, *Ok* and *Found*, only happen in these cases.

Curiously, in some cases, instead of getting the right result, it is possible to get the exactly path where the HTML or PHP files are located, the *DocumentRoot* of Apache, something like each parameter is a folder of path. The *DocumentRoot* is */home/master/www*, and are shown to the user at the Listings 45 and 46. This can be a serious security fault and it happened with, for example, operators MIEB_16 and MIFS_173, represented in Listings 50 and 49, respectively.

In another case, it is possible to view part of the rewriting rule of *mod_rewrite*, as can be seen in the Listing 39. Furthermore, there are two situations, applying the patches MIA_241 and MIA_257 (represented in Listings 51 and 52) where it is possible to see the page of php code in question, rather than the result of execution, as in the example represented in the Listing 5.2.

```
1 <?php
2
3 // Show all information, defaults to INFO_ALL
4 phpinfo();
5
6 ?>
```

Listing 5.2: *Behavior: PHPInfo - Wrong.*

This suggests that a problem has occurred during the loading of the PHP module. Thereby showing a single fault introduced into Apache module affects the operation of other, as is the case of PHP module.

6 Conclusion

In this chapter, the main conclusions regarding the work performed during this MSc will be described. Furthermore, it will also point out some possible future research directions.

6.1 Summary

The Cloud is a new model that allows to deliver on-demand IT services. The usage of this model is increasing, because it provides several advantages to the users and organizations, such as accommodation of larger loads only by adding resources. Lower costs is another significant benefit, considering that, for example, when an organization transfers an application to the cloud, they also do not have anymore to pay the fixed costs of having an IT infrastructure. However, the Cloud is not free from disturbances. Therefore, the ability of a critical system to handle bugs must be verified. This verification requires the use of fault injector tools, which are created on the software or hardware level. The development of a fault injector tool based on software level is presented in the current dissertation to contribute to the dependability and robustness evaluation of cloud systems.

This fault injector, named BugTor, is able to inject thirteen of the most representative fault types at source code (according to the specification of Durães & Madeira [Durães and Madeira, 2006]). This tool has been developed using the CDT Plugin, which has some interesting characteristics to manipulate source code, create an AST from source and change the nodes.

However, the use of this Plugin presents some limitations, for example the macros are not mapped in the created tree. Therefore, the code expanded through the macros are unreadable to the CDT. Nevertheless, there are two ways to solve this problem: solve the macros before inject faults, using GCC, or reduce the amount of code. If none of these options are feasible, it was created a workaround allowing the replacement of macros. Using this workaround, the faults are not injected into the expanded code of macros.

The development of this project required facing several challenges related to the decisions taken, such as the use of reflection to obtain the modifications to perform in the code of the tree (which later became native code), as well as the problem of macros, described above.

In the experiments performed to demonstrate the tool, faults are injected in Apache Web Server, which is one of the most used Web Server. The faults are injected through the application of each one of the most representative operators, in the module *mod_rewrite* of Apache. This module has the greater McCabe complexity in the source code of Apache Web Server.

Nevertheless, the experiments done can be considered satisfactory. The results of the performed tests with Apache show that the tool actually works, which is useful for future testing in Cloud environments. The injected faults induced Apache to behave anomalous in 213 times, from the total of 1474 experiments, equivalent to 15%. These experiments presented some problems that occur in Apache, such as the information related to the *DocumentRoot* path or the code of web page, instead of the result of its execution. These problems, among others (the use of old versions, combined with the security vulnerabilities [10]) caused a reduction in the Apache usage rate in recent years, although it is still the most widely used.

Finally, the outcome of this work is a fault injector ready for further testing with other applications in cloud environments. The use of this tool revealed some Apache Web Server problems regarding the way it reacts in the presence of faults. Even considering that the environment where the tests were performed is similar to a real cloud environment, it would be difficult to proceed with a full assessment of the cloud system.

6.2 Future Work

This work was based mainly in the creation of a SFI. Therefore, the number of experiments carried out was relatively low. Thus, in the future, it is important to carry out experiments in different scenarios, such as:

- **Two virtual machines**, one to repeat the performed tests with the presence of faults, and other without faults to verify if the isolation between virtual machines is not affected, represented at Figure 13. The goal of this experiment is to evaluate whether through fault injection in one of the virtual machines, the others are affected, checking if there was propagation of failures;

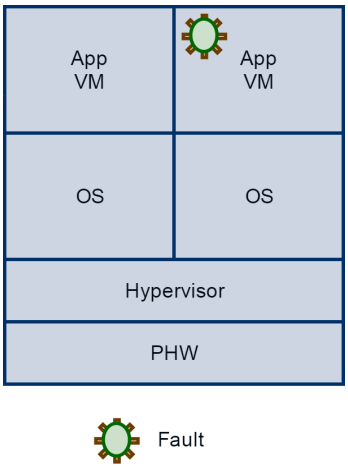

Figure 13: *Experiment.*

- **Another type of hypervisor**, native-hypervisor, instead of hosted-hypervisor, to check if the obtained results are the same;
- **Faults injected in the hypervisor**, to verify its behavior in the presence of software faults;
- A **real cloud** service provider.

Moreover, as mentioned in this dissertation, macros have been one of the main limitations encountered. Therefore, it would be pertinent and interesting to develop a mechanism that allows the reading and editing of macros in CDT Plugin.

7 References

[Ahmed, 2009] Ahmed, A. (2009). *Software Testing as a Service*. CRC Press.

[Armbrust et al., 2010] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A view of cloud computing. *Commun. ACM*, 53(4):50–58.

[Avižienis et al., 2004] Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33.

[Bridge and Miller, 1998] Bridge, N. and Miller, C. (1998). Orthogonal defect classification using defect data to improve software development. *Software Quality*, 3(1):1–8.

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

[Cerveira, 2015] Cerveira, F. (2015). Benchmarking de infraestruturas de virtualização para a cloud. *Thesis*, pages 0–53.

[Cerveira et al., 2015] Cerveira, F., Barbosa, R., Madeira, H., and Araujo, F. (2015). Recovery for virtualized environments.

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

[Cotroneo, 2013] Cotroneo, D. (2013). *Innovative Technologies for Dependable OTS-Based Critical Systems: Challenges and Achievements of the CRITICAL STEP Project*. Springer Publishing Company, Incorporated.

[De Florio, 2012] De Florio, V. (2012). *Technological Innovations in Adaptive and Dependable Systems: Advancing Models and Concepts: Advancing Models and Concepts*. Premier reference source. Information Science Reference.

[Diez and Silva, 2014] Diez, O. and Silva, A. (2014). Resilience of cloud computing in critical systems. *Quality and Reliability Engineering International*, 30(3):397–412.

[Durães, 2005] Durães, J. A. (2005). Faultloads baseadas em falhas de software para testes padronizados de confiabilidade. *Thesis*, pages 0–269.

[Durães and Madeira, 2006] Durães, J. A. and Madeira, H. S. (2006). Emulation of software faults: A field data study and a practical approach. *Software Engineering, IEEE Transactions on*, 32(11):849–867.

[Fornaeus, 2010] Fornaeus, J. (2010). Device hypervisors. In *Proceedings of the 47th Design Automation Conference*, pages 114–119. ACM.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

[ISO/IEC/IEEE 24765, 2010] ISO/IEC/IEEE 24765 (2010). Systems and software engineering – vocabulary.

[Jhawar et al., 2012] Jhawar, R., Piuri, V., and Santambrogio, M. (2012). A comprehensive conceptual system-level approach to fault tolerance in cloud computing. In *Systems Conference (SysCon), 2012 IEEE International*, pages 1–5. IEEE.

[Koopman et al., 1997] Koopman, P., Sung, J., Dingman, C., Siewiorek, D., and Marz, T. (1997). Comparing operating systems using robustness benchmarks. In *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on*, pages 72–79. IEEE.

[Madeira et al., 2000] Madeira, H., Costa, D., and Vieira, M. (2000). On the emulation of software faults by software fault injection. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 417–426. IEEE.

[Madeira et al., 2015] Madeira, H., Natella, R., and Cotroneo, D. (2015). Assessing dependability with software fault injection: A survey. volume 48.

[Martins et al., 2002] Martins, E., Rubira, C. M., and Leme, N. G. (2002). Jaca: A reflective fault injection tool based on patterns. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 483–487. IEEE.

[McCabe, 1976] McCabe, T. J. (1976). A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320.

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

[Natella, 2011] Natella, R. (2011). Achieving representative faultloads in software fault injection.

[Nindel-Edwards and Steinke, 2014] Nindel-Edwards, J. and Steinke, G. (2014). Ethical issues in the software quality assurance function. *Communications of the IIMA*, 8(1):6.

[Popek and Goldberg, 1974] Popek, G. J. and Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421.

[Regina et al., 2003] Regina, L., Martins, E., et al. (2003). Jaca—a software fault injection tool. page 667. IEEE.

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

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

[Sommerville, 2006] Sommerville, I. (2006). *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Vieira and Madeira, 2009] Vieira, M. and Madeira, H. (2009). From performance to dependability benchmarking: a mandatory path. In *Performance Evaluation and Benchmarking*, pages 67–83. Springer.

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

[Wong et al., 1997] Wong, W. E., Horgan, J. R., London, S., and Agrawal, H. (1997). A study of effective regression testing in practice. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pages 264–274. IEEE.

8 Webography

[1] "Microsoft azure," Microsoft, @15-January-2016. [Online]. Available: https://azure.microsoft.com/pt-pt/

[2] "Amazon ec2," Amazon, @15-January-2016. [Online]. Available: https://aws.amazon.com/ec2/

[3] "Google cloud," Google, @15-January-2016. [Online]. Available: https://cloud.google.com//

[4] "Difference between different Cloud services," @15-January-2016. [Online]. Available: http://www.hanusoftware.com/azurezone/whats-the-difference-between-different-cloud-services-like-iaas-paas-and-saas/

[5] VMware, "Vmware workstation pro," 2016, @15-January-2016. [Online]. Available: http://www.vmware.com/products/workstation

[6] "Virtualbox.org," @15-January-2016. [Online]. Available: http://www.virtualbox.org

[7] "Qemu - open source processor emulator," @15-January-2016. [Online]. Available: http://wiki.qemu.org/Main_Page

[8] NetCraft, "December 2015 - web server survey," @15-January-2016. [Online]. Available: http://news.netcraft.com/archives/2015/12/31/december-2015-web-server-survey.html

[9] T. Mättig, "My php performance benchmarks," @22-January-2016. [Online]. Available: http://maettig.com/code/php/php-performance-benchmarks.php

[10] NetCraft, "Are there really lots of vulnerable apache web servers?" @27-January-2016. [Online]. Available: http://news.netcraft.com/archives/2014/02/07/are-there-really-lots-of-vulnerable-apache-web-servers.html

[11] "Gs managing your thesis," @15-January-2016. [Online]. Available: https://www.scribd.com/doc/243445171/GS-ManagingYourThesis

Appendices

A Gantt diagrams

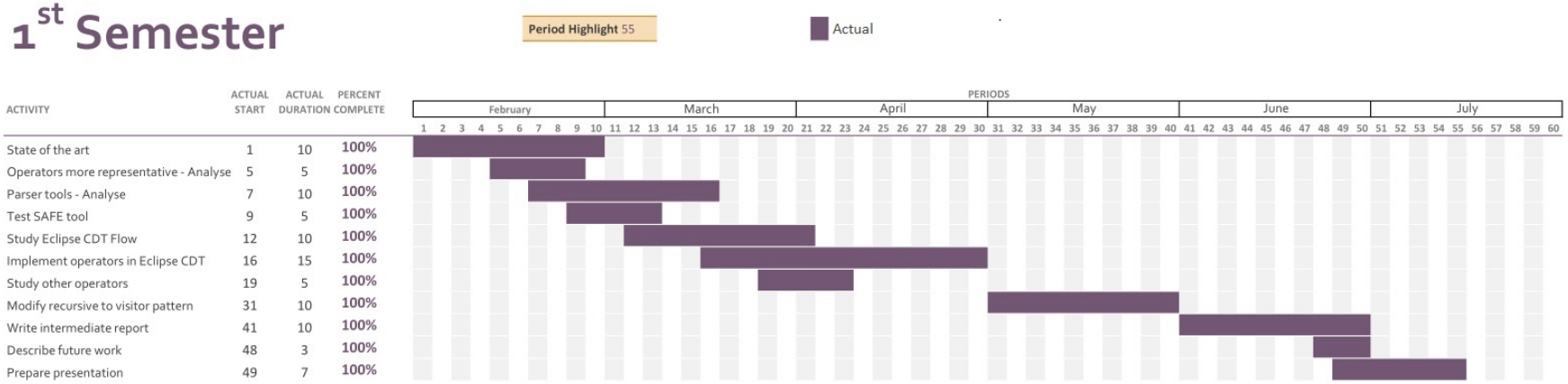

Figure 14: *First semester Gantt.*

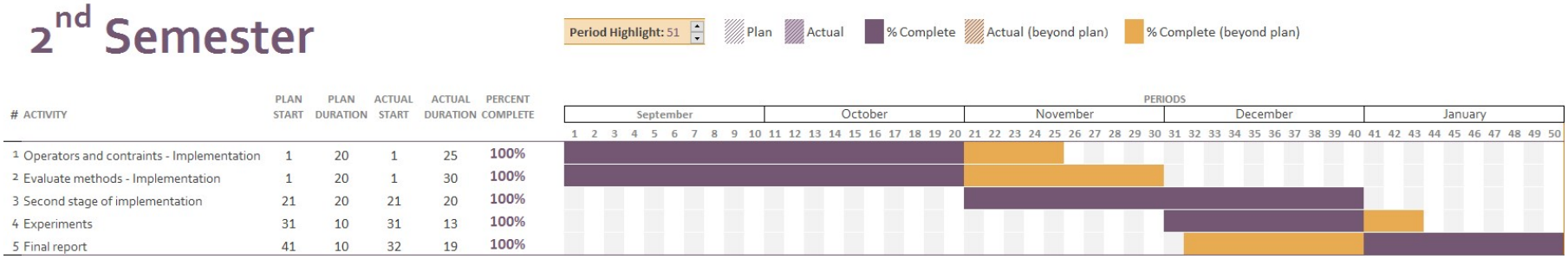

Figure 15: *Second semester Gantt.*

B Risks table

Risc Area	Preventive 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 16: *Risks (Adapted from Managing Your Thesis: a Quick Reference Guide by Curtin University [11]).*

C Constraints - examples

Constraint C01 - Return value of the function must not be used

- Related with the operator: **MFC**

```
1 (...)
2
3 i = function();
4
5 (...)
```

Listing 1: *Constraint example: C01 - False.*

```
1 (...)
2
3 function();
4
5 (...)
```

Listing 2: *Constraint example: C01 - True.*

Constraint C02 - Call/Assignment/The if construct/The statements must not be the only statement in the block

- Related with the operators: **MIFS**, **MFC**, **MLPA**, **MVAE**, **MVAV**, **MVIV**

```
1 (...)
2 {
3         i = 1;
4 }
5
6 (...)
```

Listing 3: *Constraint example: C02 - False.*

```
1 (...)
2 {
3         i = 1;
4         i = function();
5 }
6 (...)
```

Listing 4: *Constraint example: C02 - True.*

Constraint C03 - Variable must be inside stack frame

- Related with the operators: **MVAE**, **MVAV**, **MVIV**, **WVAV**, **WPFV**

```
1 (...)
2
3         int i = 0;
4
5 (...)
```

Listing 5: *Constraint example: C03 - False.*

```
1 (...)
2 {
3         int i = 1;
4 }
5 (...)
```

Listing 6: *Constraint example: C03 - True.*

Constraint C04 - Must be the first assignment for that variable in the module

- Related with the operators: **MVIV**, **WVAV**

```
1 (...)
2
3 int i = 0;
4 i = 20;
5
6 (...)
```

Listing 7: *Constraint example: C04 - False.*

```
1 (...)
2
3 int i;
4 i = 0;
5
6 (...)
```

Listing 8: *Constraint example: C04 - True.*

```
1 (...)
2
3 int i = 0;
4
5 (...)
```

Listing 9: *Constraint example: C04 - True.*

Constraint C05 - Assignment must not be inside a loop

- Related with the operator: **MVIV**

```
1 (...)
2 while(i < 10){
3         int j = 0;
4 }
5 (...)
```

Listing 10: *Constraint example: C05 - False.*

```
1 (...)
2 int function() {
3         int i = 0;
4 }
5 (...)
```

Listing 11: *Constraint example: C05 - True.*

```
1 (...)
2 do {
3         int j = 0;
4 } while(i < 10);
5 (...)
```

Listing 12: *Constraint example: C05 - False.*

```
1 (...)
2 for(int i = 0; i < 10 ; i++) {
3         int j = 0;
4 }
5 (...)
```

Listing 13: *Constraint example: C05 - False.*

Constraint C06 - Assignment must not be part of a for construct

- Related with the operators: **WVAV**, **MVIV**, **MVAV**, **MVAE**

```
1 (...)
2
3 int i = 0;
4
5 (...)
```

Listing 14: *Constraint example: C06 - False.*

```
1 (...)
2 for(int i = 0; i < 10 ; i++) {
3         (...)
4 }
5 (...)
```

Listing 15: *Constraint example: C06 - True.*

Constraint C07 - Must not be the first assignment for that variable in the module

- Related with the operators: **MVAE**, **MVAV**

Is the negation of constraint **C04**.

Constraint C08 - The if construct must not be associated to an else construct

- Related with the operators: **MIFS**, **MIA**

```
1 (...)
2 if (i < 10) {
3
4 } else {
5
6 }
7 (...)
```

Listing 16: *Constraint example: C08 - False.*

```
1 (...)
2 if (i < 10) {
3
4 }
5 i = 20;
6
7 (...)
```

Listing 17: *Constraint example: C08 - True.*

Constraint C09 - Statements must not include more than five statements and not include loops

- Related with the operators: **MIFS**, **MIA**

```
1 (...)
2 int i = 0;
3 while(j < 10) {
4         i++;
5 }
6 (...)
```

Listing 18: *Constraint example: C09 - False.*

```
1 (...)
2 int i = 0;
3 int j = 10;
4 int counter = 0;
5 (...)
```

Listing 19: *Constraint example: C09 - True.*

Constraint C10 - Statements are in the same block, do not include more than five statements, nor loops

- Related with the operator: **MLPA**

```
1 (...)
2 {
3         int i = 0;
4         while(j < 10) {
5                 i++;
6         }
7 }
8 (...)
```

Listing 20: *Constraint example: C10 - False.*

```
1 (...)
2 {
3         int i = 0;
4         int j = 10;
5         int counter = 0;
6 }
7 (...)
```

Listing 21: *Constraint example: C10 - True.*

Constraint C11 - There must be at least two variables in this module

- Related with the operator: **WPFV**

```
1 (...)
2 int function()
3 {
4         int i = 0;
5         while( i < 10) print(i);
6 }
7 (...)
```

Listing 22: *Constraint example: C11 - False.*

```
1 (...)
2 int function()
3 {
4         int i = 0;
5         int j = 0;
6         return calc(i,j);
7 }
8 (...)
```

Listing 23: *Constraint example: C11 - True.*

Constraint C12 - Must have at least two branch conditions

- Related with the operators: **MLAC**, **MLOC**

```
1 (...)
2 if (j < 50) {
3
4 }
5 (...)
```

Listing 24: *Constraint example: C12 - False.*

```
1 (...)
2 if (i < 10 || i > 20) {
3
4 }
5 (...)
```

Listing 25: *Constraint example: C12 - True.*

```
1 (...)
2 while (j < 50) {
3
4 }
5 (...)
```

Listing 26: *Constraint example: C12 - False.*

```
1 (...)
2 if (i < 10 || i > 20 && j < 50) {
3
4 }
5 (...)
```

Listing 27: *Constraint example: C12 - True.*

Constraint C13 - The if construct must be associated to an else construct

- Related with the operator: **MIEB**

Is the negation of constraint **C08**.

D Macros

```
#include <AL/al.h>
#include <AL/alc.h>
#include <AL/alut.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string>
#include <string.h>
#include "RgbImage.h"
#include <sstream>
#include <string>
#include <iostream>
#include <vector>  //for std::vector
#include "materiais.h"

#define BLUE 0.0, 0.0, 1.0, 1.0
#define RED 1.0, 0.0, 0.0, 1.0
#define GREEN 0.0, 1.0, 0.0, 1.0
#define WHITE 1.0, 1.0, 1.0, 1.0
#define BLACK 0.0, 0.0, 0.0, 1.0

#define NUM_PARTICLES     1000          /* Number of particles   */
#define NUM_DEBRIS        70            /* Number of debris      */
#define NUM_PARTICLES_RAIN 10000

void init( ) {
    int x, z;

    glEnable(GL_FOG);
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClearDepth(1.0);
    glEnable(GL_DEPTH_TEST);

    for (z = 0; z < 21; z++) {
        for (x = 0; x < 21; x++) {
            ground_points[x][z][0] = x - 10.0;
            ground_points[x][z][1] = accum;
            ground_points[x][z][2] = z - 10.0;

            ground_colors[z][x][0] = r; // red value
            ground_colors[z][x][1] = g; // green value
            ground_colors[z][x][2] = b; // blue value
            ground_colors[z][x][3] = 0.0; // acummulation factor
        }
    }
}

(...)
```

Listing 28: *Example of code with macros in the beginning.*

```
static char* lookup_map_program(request_rec* r, apr_file_t* fpin, apr_file_t*
    ↪ fpout, char* key)
{
char* buf;
char c;
apr_size_t i, nbytes, combined_len = 0;
apr_status_t rv;
const char* eol = APR_EOL_STR;
apr_size_t eolc = 0;
int found_nl = 0;
result_list *buflist = NULL, *curbuf = NULL;
#ifndef NO_WRITEV
struct iovec iova[2];
apr_size_t niov;
#endif
if (fpin == NULL || fpout == NULL || ap_strchr(key, '\n')) {
    return NULL;
}
if (rewrite_mapr_lock_acquire) {
    rv = apr_global_mutex_lock(rewrite_mapr_lock_acquire);
    if (rv != APR_SUCCESS) {
        ap_log_rerror(APLOG_MARK, APLOG_ERR, rv, r, APLOGNO(00659)
            "apr_global_mutex_lock(rewrite_mapr_lock_acquire) "
            "failed");
        return NULL;
    }
}
#ifdef NO_WRITEV
    nbytes = strlen(key);
    apr_file_write_full(fpin, key, nbytes, NULL);
    nbytes = 1;
    apr_file_write_full(fpin, "\n", nbytes, NULL);
#else
iova[0].iov_base = key;
iova[0].iov_len = strlen(key);
iova[1].iov_base = "\n";
iova[1].iov_len = 1;
niov = 2;
apr_file_writev_full(fpin, iova, niov, &nbytes);
#endif
buf = apr_palloc(r->pool, REWRITE_PRG_MAP_BUF + 1);
nbytes = 1;
apr_file_read(fpout, &c, &nbytes);
do{
    i = 0;
    while (nbytes == 1 && (i < REWRITE_PRG_MAP_BUF)){
        if (c == eol[eolc]) {
            if (!eol[++eolc]) {
                --eolc;
                if (i < eolc) {
                    curbuf->len -= eolc - i;
                    i = 0;
                }else{
                    i -= eolc;
                }
                ++found_nl;
                break;
            }
        }else
            if (eolc) {
                eolc = 0;
            }else
                if (c == '\n') {
                    ++found_nl;
                    break;
                }

        buf[i++] = c;
        apr_file_read(fpout, &c, &nbytes);
    }
```

```
70      if (buflist || (nbytes == 1 && !found_nl)) {
71          if (!buflist) {
72              curbuf = buflist = apr_palloc(r->pool, sizeof (*buflist));
73          }else
74              if (i) {
75                  curbuf->next = apr_palloc(r->pool, sizeof (*buflist));
76                  curbuf = curbuf->next;
77              }
78
79          curbuf->next = NULL;
80          if (i) {
81              curbuf->string = buf;
82              curbuf->len = i;
83              combined_len += i;
84              buf = apr_palloc(r->pool, REWRITE_PRG_MAP_BUF);
85          }
86          if (nbytes == 1 && !found_nl) {
87              continue;
88          }
89      }
90      break;
91  } while (1);
92  if (buflist) {
93      char* p;
94      p = buf = apr_palloc(r->pool, combined_len + 1);
95      while (buflist){
96          if (buflist->len) {
97              memcpy(p, buflist->string, buflist->len);
98              p += buflist->len;
99          }
100         buflist = buflist->next;
101     }
102     *p = '\0';
103     i = combined_len;
104 }else{
105     buf[i] = '\0';
106 }
107 if (rewrite_mapr_lock_acquire) {
108     rv = apr_global_mutex_unlock(rewrite_mapr_lock_acquire);
109     if (rv != APR_SUCCESS) {
110         ap_log_rerror(APLOG_MARK, APLOG_ERR, rv, r, APLOGNO(00660)
111             "apr_global_mutex_unlock(rewrite_mapr_lock_acquire) "
112             "failed");
113         return NULL;
114     }
115 }
116 if (i == 4 && !strcasecmp(buf, "NULL")) {
117     return NULL;
118 }
119 return buf;
120 }
```

Listing 29: *Part of mod_rewrite source code. Function with embedded macros ifndef,ifdef and else.*

E Experiments

```
#!/bin/bash

ls patchs/ | grep patch > files.txt
for f in $(cat files.txt)
do
        cp patchs/mod_rewrite.c mod_rewrite.c
        cp patchs/$f $f

        date >> log.txt
        echo "apply patch $f" >> log.txt
        patch -f < "$f"
        timeout -sKILL 30m apxs -c -i mod_rewrite.c

        /etc/init.d/apache2 restart

        ### Apache Test
        timeout -sKILL 10m w3m -dump 127.0.0.1/index.html > "$f.$(date +%Y-%m-%
            ↪ d_%H-%M).apache"
        ### Mod_Rewrite Test
        timeout -sKILL 10m w3m -dump 127.0.0.1/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
            ↪ abcdefghijklmnopqrstuvwxyz/0123456789/%C3%8B%27%E2%80%9D%C3%87%
            ↪ E2%80%B9%C3%AE%C3%9D%20%C2%A5%C2%B6%C5%92%C3%95%E2%80%9C%C3%B1%
            ↪ C3%BF%C2%BA%C3%B2%C3%9A%C3%A5%3C%CB%86 > "$f.$(date +%Y-%m-%d_%H
            ↪ -%M).out"
        ### PHP Test
        timeout -sKILL 10m w3m -dump 127.0.0.1/phpinfo.php > "$f.$(date +%Y-%m
            ↪ -%d_%H-%M).phpinfo"

        ### PHP BENCHMARK
        timeout -sKILL 10m w3m -dump 127.0.0.1/php-performance-benchmarks.php >
            ↪  "$f.$(date +%Y-%m-%d_%H-%M).phpbench"

        ### System Test
        uname -a > "$f.system"
        echo "ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/0123456789"
            ↪  >> "$f.system"
        uptime >> "$f.system"
        df -h >> "$f.system"
        dmesg >> "$f.system"

        ### Apache Benchmark
        timeout -sKILL 10m ab -n 10 -c 2 -g "$f.$(date +%Y-%m-%d_%H-%M).log"
            ↪ 127.0.0.1/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz
            ↪ /0123456789/%C3%8B%27%E2%80%9D%C3%87%E2%80%B9%C3%AE%C3%9D%20%C2%
            ↪ A5%C2%B6%C5%92%C3%95%E2%80%9C%C3%B1%C3%BF%C2%BA%C3%B2%C3%9A%C3%
            ↪ A5%3C%CB%86

        patch -R -f < $f
        rm $f
        date >> log.txt
        echo "another experience"  >> log.txt
done
```

Listing 30: *Bash script of automated experiments.*

F Behaviors - examples

```
1
2 Array ( [0] => ABCDEFGHIJKLMNOPQRSTUVWXYZ [1] => abcdefghijklmnopqrstuvwxyz [2]
3 => 0123456789 [3] => Ë”’‹ÇîÝ Œ“¥¶Õñÿº òÚåˆ< )
```

Listing 31: *Behavior: Correct.*

```
1 Bad Request
2
3 Your browser sent a request that this server could not understand.
4
5
6
7 Apache/2.4.12 (Ubuntu) Server at 127.0.0.1 Port 80
```

Listing 32: *Behavior: Bad request.*

```
1 Forbidden
2
3 You don't have permission to access /ABCDEFGHIJKLMNOPQRSTUVWXYZ/
4 abcdefghijklmnopqrstuvwxyz/0123456789/Ë”’‹ÇîÝ Œ“¥¶Õñÿº òÚåˆ< on this server.
5
6
7
8 Apache/2.4.12 (Ubuntu) Server at 127.0.0.1 Port 80
```

Listing 33: *Behavior: Forbidden.*

```
1 Found
2
3 The document has moved here.
4
5
6
7 Apache/2.4.12 (Ubuntu) Server at 127.0.0.1 Port 80
```

Listing 34: *Behavior: Found.*

```
1  Internal Server Error
2
3  The server encountered an internal error or misconfiguration and was unable to
4  complete your request.
5
6  Please contact the server administrator at webmaster@localhost to inform them
7  of the time this error occurred, and the actions you performed just before this
8  error.
9
10 More information about this error may be available in the server error log.
11
12
13
14 Apache/2.4.12 (Ubuntu) Server at 127.0.0.1 Port 80
```

Listing 35: *Behavior: Internal Server Error.*

```
1 Not Found
2
3 The requested URL /ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/
4 0123456789/Ë”’‹ÇîÝ Œ“¥¶Õñÿº òÚåˆ< was not found on this server.
5
6
7
8 Apache/2.4.12 (Ubuntu) Server at 127.0.0.1 Port 80
```

Listing 36: *Behavior: Not found - url OK.*

```
Not Found

The requested URL /  s    was not found on this server.



Apache/2.4.12 (Ubuntu) Server at 127.0.0.1 Port 80
```

Listing 37: *Behavior: Not found - wrong url.*

```
OK

The server encountered an internal error or misconfiguration and was unable to
complete your request.

Please contact the server administrator at webmaster@localhost to inform them
of the time this error occurred, and the actions you performed just before this
error.

More information about this error may be available in the server error log.



Apache/2.4.12 (Ubuntu) Server at 127.0.0.1 Port 80
```

Listing 38: *Behavior: Apache error - Ok.*

```

Array ( [0] => ABCDEFGHIJKLMNOPQRSTUVWXYZ [1] => abcdefghijklmnopqrstuvwxyz [2]
=> 0123456789 [3] => Ë”’‹ÇîÝ Œ“¥¶Õñÿº òÚåˆ< [QSA,L] )
```

Listing 39: *Behavior: Wrong output - behavior 1.*

```

Array ( [0] => [1] => ABCDEFGHIJKLMNOPQRSTUVWXYZ [2] =>
abcdefghijklmnopqrstuvwxyz [3] => 0123456789 [4] => Ë”’‹ÇîÝ Œ“¥¶Õñÿº òÚåˆ< [5]
=> abcdefghijklmnopqrstuvwxyz [6] => 0123456789 [7] => Ë”’‹ÇîÝ Œ“¥¶Õñÿº òÚåˆ< )
```

Listing 40: *Behavior: Wrong output - behavior 2.*

```

Array ( [0] => ABCDEFGHIJKLMNOPQRSTUVWXYZ [1] => abcdefghijklmnopqrstuvwxyz [2]
=> 0123456789 [3] => Ë”’‹ÇîÝ Œ“¥¶Õñÿº òÚåˆ<??url=ABCDEFGHIJKLMNOPQRSTUVWXYZ [4]
=> abcdefghijklmnopqrstuvwxyz [5] => 0123456789 [6] => Ë”’‹ÇîÝ Œ“¥¶Õñÿº òÚåˆ< )
```

Listing 41: *Behavior: Wrong output - behavior 3.*

```

Array ( [0] => ABCDEFGHIJKLMNOPQRSTUVWXYZ )
```

Listing 42: *Behavior: Wrong output - behavior 4.*

```

Array ( [0] => $1 )
```

Listing 43: *Behavior: Wrong output - behavior 5.*

```

Array ( [0] => )
```

Listing 44: *Behavior: Wrong output - behavior 6.*

```

Array ( [0] => [1] => home [2] => master [3] => www [4] =>
ABCDEFGHIJKLMNOPQRSTUVWXYZ [5] => abcdefghijklmnopqrstuvwxyz [6] => 0123456789
[7] => Ë”’‹ÇîÝ Œ“¥¶Õñÿº òÚåˆ< )
```

Listing 45: *Behavior: Wrong output - behavior 7.*

```
1
2 Array ( [0] => [1] => home [2] => master [3] => www [4] =>
3 ABCDEFGHIJKLMNOPQRSTUVWXYZ )
```

Listing 46: *Behavior: Wrong output - behavior 8.*

```
1
2 Array ( [0] =>          )
```

Listing 47: *Behavior: Wrong output - behavior 9.*

```
1
2 Array ( [0] =>    C     )
```

Listing 48: *Behavior: Wrong output - behavior 10.*

G Patches - examples

```
--- mod_rewrite.c
+++ mod_rewrite.c._MIFS_173     2015-12-16 19:14:05.769023500 +0000
@@ -2810,11 +2810,6 @@
                         ctx->uri, ctx->uri, r->path_info));
     ctx->uri = apr_pstrcat(r->pool, ctx->uri, r->path_info, NULL);
  }
-if (!is_proxyreq && strlen(ctx->uri) >= dirlen && !strncmp(ctx->uri, ctx->
    ↪ perdir, dirlen)) {
-    rewritelog((r, 3, ctx->perdir, "strip per-dir prefix: %s -> %s",
-                        ctx->uri, ctx->uri + dirlen));
-    ctx->uri = ctx->uri + dirlen;
-}
  }
  rewritelog((r, 3, ctx->perdir, "applying pattern '%s' to uri '%s'",
                  p->pattern, ctx->uri));
```

Listing 49: *Patch: _MIFS_173.*

```
--- mod_rewrite.c
+++ mod_rewrite.c._MIEB_16      2015-12-16 19:11:36.805336700 +0000
@@ -1902,9 +1902,7 @@
  if (path == NULL) {
     a->directory = NULL;
  }else{
-    if (path[strlen(path) - 1] == '/') {
-        a->directory = apr_pstrdup(p, path);
-    }else{
+    {
         a->directory = apr_pstrcat(p, path, "/", NULL);
     }
  }
```

Listing 50: *Patch: _MIEB_16.*

```
--- mod_rewrite.c
+++ mod_rewrite.c._MIA_241      2015-12-16 19:15:19.070573400 +0000
@@ -3245,7 +3245,7 @@
  }
  ofilename = r->filename;
  oargs = r->args;
-if (r->filename == NULL) {
+{
  r->filename = apr_pstrdup(r->pool, r->uri);
  rewritelog((r, 2, dconf->directory, "init rewrite engine with"
                      " requested uri %s", r->filename));
```

Listing 51: *Patch: _MIA_241.*

```
--- mod_rewrite.c
+++ mod_rewrite.c._MIA_257      2015-12-16 19:15:32.692489000 +0000
@@ -3362,7 +3362,7 @@
  ap_set_content_type(r, t);
  }
  t = apr_table_get(r->notes, REWRITE_FORCED_HANDLER_NOTEVAR);
-if (t && *t) {
+{
  rewritelog((r, 1, NULL, "force filename %s to have the "
                      "Content-handler '%s'", r->filename, t));
  r->handler = t;
```

Listing 52: *Patch: _MIA_257.*