Faculty of Sciences and Technology
of the University of Coimbra

Informatics Engineering

2014/2015

Thesis Report

Benchmarking de Infraestruturas de Virtualização para a Cloud

Frederico Cerveira
fmduarte@dei.uc.pt

Orientadores:
Raul Barbosa
Henrique Madeira

6th July 2015

Faculty of Sciences and Technology
of the University of Coimbra

Informatics Engineering

2014/2015

Thesis Report

Benchmarking de Infraestruturas de Virtualização para a Cloud

Frederico Cerveira
fmduarte@dei.uc.pt

Orientadores:
Raul Barbosa
Henrique Madeira

Juri Arguente: Mario Rela
Juri Vogal: Jorge Granjal

6th July 2015

Resumo

O *Cloud Computing* tem sofrido uma enorme expansão nos últimos anos, com cada vez mais organizações a migrar os seus sistemas, que se encontravam previamente *in-house*, para a *Cloud*. No entanto esta migração ainda é vista com olhos duvidosos por algumas companhias, com medo de confiar os seus sistemas a entidades externas sobre as quais não possuem controlo. Uma das razões desta falta de confiança deve-se à problemática da resiliência dos seus sistemas. Este dilema lança as fundações para esta tese. O nosso trabalho define as bases para o desenvolvimento de uma *benchmark* de resiliência para sistemas de *Cloud Computing*, apresenta um mecanismo *watchdog* capaz de detectar problemas e restaurar o serviço de um sistema virtualizado para *Cloud*, e relata os resultados obtidos de várias experiências. O bem mais valioso produzido durante esta tese foram as 3 ferramentas para *Fault Injection*, e que a partir de agora estão disponíveis para serem usadas em futuros projectos de investigação.

Abstract

Cloud computing has witnessed a tantalizing growth in recent years, with more and more companies migrating their previously in-house systems to the Cloud, and Cloud Providers racing to match this growth by building newer and bigger datacenters. Nonetheless, this migration is still seen with dubious eyes by some companies, rightfully reticent in confiding their systems to external entities, over whom they possess little to no control. One of the reasons of this distrust has to do with the resilience of their systems. This dilemma set the foundation for this thesis. Our work sets the foundation for the development of a resilience benchmark for Cloud Computing systems, presents a watchdog mechanism capable of detecting problems and recovering the service of virtualized cloud systems and reports the results of various campaigns. The most important asset produced from this thesis are the 3 Fault Injection tools that we developed and from now on can support future research projects.

Keywords. Fault Injection, Virtualization, Cloud Computing, Xen, Benchmarking

Contents

List of Figures

List of Tables

List of Acronyms

FI	**F**ault **I**njection
SWIFI	**S**oft**w**are **I**mplemented **F**ault **I**njection
VT-d	**V**irtualization **T**echnology for **D**irected I/O
PV	**P**ara**v**irtualization
HVM	**H**ardware **V**irtual **M**achine
EPT	**E**xtended **P**age **T**able
VM	**V**irtual **M**achine
KVM	**K**ernel-based **V**irtual **M**achine
PRNG	**P**seudo-**R**andom **N**umber **G**enerator
SHA1	**S**ecure **H**ash **A**lgorithm Version **1**
HTTP	**H**ypertext **T**ransport **P**rotocol
SSD	**S**olid-**S**tate **D**rive
HDD	**H**ard **D**isk **D**rive
LVM	**L**ogical **V**olume **M**anager
TCP	**T**ransmission **C**ontrol **P**rotocol
HTML	**H**yper**T**ext **M**arkup **L**anguage

1 Introduction

The work here presented has been carried under *"Dissertação / Estágio"* of *"Mestrado em Engenharia Informática"* of *Universidade de Coimbra*, and took place in CISUC (Center for Informatics and Systems of the University of Coimbra). Furthermore, this work has been supported by a fellowship provided under the DECAF project, whose aim is the analysis of the behaviour of virtualized Cloud Computing infrastructure in the presence of faults.

1.1 Motivation

Cloud Computing is growing in popularity compared to traditional computing. In order to accommodate for this increased interest cloud providers are building data centers all around the globe. These data centers feature massive amounts of commodity hardware, which was not designed to endure this kind of use and is often undervolted in order to reduce energy consumption and cooling requirements [6]. Furthermore, cloud providers strive to maintain their hardware working always at full capacity, by having multiple VMs in the same physical node. This allows the remaining systems to be powered down. All these factors lead to an increase in the probability of faults in the hardware [39][37].

On the other hand, with the increasing popularity of cloud systems, companies are migrating more mission-critical systems from private datacenters to cloud computing.

These reasons make it important to study the behaviour of a virtualized cloud system under the influence of soft-errors. This information can then be used for many useful purposes, such as to propose mechanisms for fault tolerance and for benchmarking of cloud infrastructures. Therefore helping to reduce the distrust between cloud providers and customers.

1.2 Objectives

The objective of this thesis was to analyze the behaviour of virtualized cloud infrastructures under the presence of faults, which would culminate with the proposal of a cloud resilience benchmarking that cloud clients and providers could use in their systems. Due to the lack of time available during the thesis, at the end of the first semester, we opted to limit our scope only to hardware faults (e.g., register and memory bitflips), and relegate software faults to future works. With the development of our work during the second semester, we decided to change the direction of our work towards a more research influenced approach, which allowed us to write two conference papers, one of which has been accepted for EDCC 2015. However this change implicated that it would be impossible for us to successfully propose a resilience benchmark standard, as was the original goal of the thesis. This change also presented us with new challenges that required the creation of new fault injection tools. The development of the fault injection tools eventually became both the main objective and area of focus of this thesis.

The choice of which hypervisor package to focus our study on was taken at the beginning of the thesis, still in the first semester, and was the result of an analysis of the available hypervisors, their features and their representativeness of real-world cloud systems. There was a wealth of options from which to choose the hypervisor, as can be seen in the Table 1, where hypervisor features are compared [13]. The final choice was to use Xen, because of the fact that its open-source license, it features the same range of features has any other top-of-the-line hypervisor and it is used in many high-grade deployments all over the world, particularly in Amazon AWS [11].

	KVM [25]	Hyper-V 2012 [43]	VirtualBox [35]	VMWare ESX [42]	Xen [41]
Type	Native	Native	Hosted	Native	Native
License	Open-Source	Commercial	Commercial	Open-Source	Open-Source
USB Support	✓	✓	✓	✓	
GUI	✓	✓	✓	✓	✓
Live Memory Allocation	✓	✓	✓	✓	✓
Snapshots	✓	✓	✓	✓	✓
Live Migration	✓	✓	✓	✓	✓

Table 1: Comparison of hypervisor packages

1.3 Document Structure

This document is structured as follows: the next chapter will present the state-of-the-art in the adjoining areas of this work, with particular attention to *Cloud Computing*, *Benchmarking*, *Fault Injection* and *Virtualization*. In Chapter 3, a detailed overview of the fault injection tools developed to provide support for the various experiments by us conducted is given. Chapter 4 presents the watchdog mechanism that we developed and which is part of the content of the article accepted in the conference EDCC 2015. This mechanism is capable of restoring the correct service of a virtualized system that is producing incorrect behaviour, even in the presence of an unresponsive system. To assess the effectiveness of this mechanism we performed a benchmark between a classic virtualized system and the same virtualized system using our watchdog. Chapter 5 presents and analyzes the results of the various experiments performed during this thesis, which consisted of injecting errors in various parts of the virtualized system (e.g., hypervisor, applications running in privileged and guest VMs) and had the objective of assessing the behaviour of a virtualized system under the presence of soft-errors. Their importance in defining the behaviour in this scenario is undeniable, given the lack of previous data in this specific area. The majority of the results here presented were also part of the paper that was accepted for the conference. Since the paper was delivered we have continued with similar experiments that have gathered further information that will be presented in this report, and later will hopefully be part of another paper that complements the results of the first one. Finally, in Chapter 6 we reflect about what was accomplished, how it can be useful for the community, how the work performed in this thesis can be expanded and what future work possibilities have risen along the way.

2 Related Work

In this chapter, a overview of the state-of-the-art in the areas of *Cloud Computing*, *Benchmarking*, *Fault Injection* and *Virtualization* is provided. Regarding the theme of this thesis, previous research is recent and scarce [7], which can indicate the recent demand for Cloud Computing and anticipate this topic as a focus of research in the following years.

In the "How is the Weather tomorrow? Towards a Benchmark for the Cloud"[7] paper, the authors argue that current benchmarks, such as the various TPC benchmarks, are not adequate for cloud computing, and present their ideas for a new benchmark that incorporates, among others, fault tolerance. The basis of their work revolves around fixing the shortcomings they deem TPC-W presents for cloud systems.

2.1 Cloud Computing

According to "The NIST definition of cloud computing"[31], 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.".

In the same publication, 5 characteristics are stated as essential to Cloud Computing.

- **On-demand self-service**: A consumer can request and obtain computing capabilities without requiring human interaction.
- **Broad network access**: Capabilities are accessible over the network through standard mechanisms.
- **Resource pooling**: The provider has a pool of resources to server multiple consumers, where the resources are dynamically assigned and reassigned according to demand. Usually the consumer has no control or knowledge over the physical location of the resources.
- **Rapid elasticity**: Resources can be elastically provisioned and released, in other to adapt to demand.
- **Measured service**: Resource usage can be monitored and controlled in a transparent way both by the provider and the consumer.

2.1.1 Service Models

There are 3 different service models that can be used for Cloud Computing.

- Software as a Service (SaaS)
- Platform as a Service (PaaS)

- Infrastructure as a Service (IaaS)

The Software as a Service model provides the customer with access through various client interfaces to software services running on the cloud. Examples of this model are Google Docs, Google Sheets and Microsoft Office 365.

In the Platform as a Service model the customer can deploy his applications into a cloud infrastructure, by using programming languages, libraries, service and tools provided by the cloud provider. The customer is unable to manage or control the underlying cloud infrastructure, keeping only control over the deployed applications. Examples are Windows Azure, Elastic Beanstalk, Apache Strato, Heroku and Google App Engine.

Finally, Infrastructure as a Service is a model where the customer is provided with the capability to manage the processing, storage, network and other resources where his applications will be run. Examples are Amazon EC2, Google Compute Engine and Rackspace.

In Figure 1, a visual comparison between Traditional IT and IaaS, SaaS, PaaS is presented.

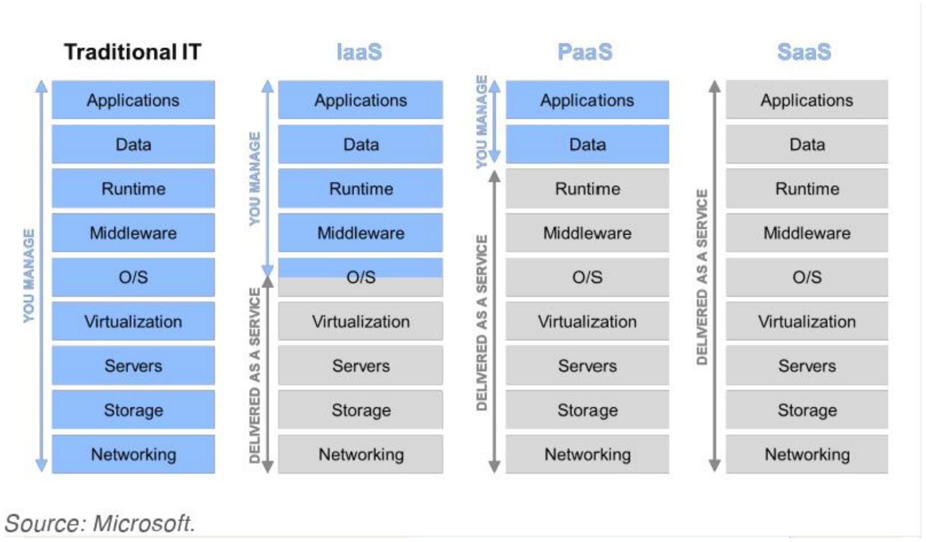

Figure 1: Comparison between Traditional IT, IaaS, SaaS and PaaS

2.1.2 Deployment Models

A cloud computing infrastructure can be deployed in a variety of different settings. Namely:

- Private Cloud
- Community Cloud
- Public Cloud

- Hybrid Cloud

In a Private Cloud, the cloud infrastructure is provisioned for exclusive use by a single organization. Whereas a Community Cloud is design to be used by a group of organizations that share similar concerns. A Public Cloud is provisioned for use by the the general public (e.g., Amazon AWS, Microsoft Azure, ...). A Hybrid Cloud possesses a cloud infrastructure that is a composed by two or more distinct cloud infrastructures (private, community, or public) that while remaining unique entities, are linked together by standardized or proprietary technology that allows data and application portability (e.g., cloud bursting for load balancing between clouds).

2.2 Benchmarking

Computer benchmarks are standardized tools that allow the evaluation and comparison of different systems or components in different areas, such as, performance, security, dependability or resilience.

In order to produce valid and useful benchmarks there is a set of criteria which must be respected. These criteria have been refined through the research in this area. One of those publications was "The Benchmark Handbook" [15], where four important criteria were defined.

- **Relevance**: the benchmark has to focus on typical and representative operations of the problem domain.
- **Portability**: the benchmark should be capable of running in several different systems and architectures.
- **Scalability**: it should be able to cover both small and large systems
- **Simplicity**: in order to avoid lack of credibility, the benchmark should strive to be clear and simple to understand.

Subsequent research [17] has yielded further criteria.

- **Relevant**: the benchmark must simulate useful and representative operations of the problem domain.
- **Repeatable**: the benchmark's results can be reproduced when running the same benchmark under similar conditions.
- **Fair&Portable**: the benchmark needs to be easily portable and do not favor or penalize one system or architecture over another, it should provide a fair comparison between systems.
- **Verifiable**: the results provided by the benchmark need to inspire confidence as of their validity and representativeness. This can be done through the review of the benchmark by external auditors.
- **Economical**: the benchmark should be affordable to run.

The most common kind of benchmarks are performance benchmarks. These benchmarks focus on evaluating the performance of a system or component for posterior comparison. Many standardization organizations and their respective standards have been created. Two of the most influential are SPEC (Standard Performance Evaluation Corporation) and TPC (Transaction Processing Performance Council).

Another kind of benchmarking are dependability benchmarks. As defined by the International Federation for Information Processing (IFIP) Working Group 10.4, dependability is "the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers" [18]. Extrapolating from this definition it is possible to state that Dependability Benchmarking is a standardized procedure used to assess dependability-related measures of a system or component. Such measures are usually stated as:

- **Availability**: being available to perform a service correctly.
- **Reliability**: continuity of the service in correct operation.
- **Safety**: absence of catastrophic consequences.
- **Confidentiality**: no unintended disclosure of information.
- **Integrity**: no improper system state alterations.
- **Maintainability**: ability of undergoing repairs and maintenance operations.

Finally, we define Resilience Benchmarking, one important part of this thesis. Resilience Benchmarking integrates concepts from performance and dependability benchmarks, and aims at providing methods for characterizing, quantifying and comparing the system behaviour in the presence of faults.

2.3 Fault Injection

Fault Injection is an important technique when assessing the behavior of a system and its fault tolerance mechanisms under the presence of faults. Fault Injection can be defined as "the process of deliberately introducing faults or errors in computer systems, allowing researchers and system designers to study how computer systems behave in the presence of faults" [4]. Fault Injection is present in many different contexts, from the automotive industry to the aerospace industry, or in the aviation industry, as well as in distributed and embedded systems.

Given the repeated use of some Fault Injection related concepts in this document, an explanation of them is provided ahead, thereby allowing the reader to perform an analysis of this document without any ambiguity. A Fault is the cause of an error. An Error is the discrepancy between the expected and the obtained result. A Failure is an event that occurs when the delivered service is unable to perform its desired function in the presence of an error.[2]

A Workload is a operation that will be used to stress the system. A Faultload is a operation that will interfere with the system. A Target System can be considered the system that will endure the testing. In this system, a Workload and a Faultload will be executing. A Control System is a system similar to the Target System, but where no Faultload will be executed. In our experiments, the purpose of this system it to assess the isolation between VMs. A Command and Control System is an external system, connected both to the Control and to the Target system, usually through a network, which performs the tasks of starting, managing and stopping the Workload and the Faultload, as well as extracting and analyzing the obtained results. A Fault Injection Experiment is the execution of a Workload and a Faultload. In each Fault Injection Experiment only one fault is injected. This will guarantee that the execution state is not corrupted by previous injected faults. This strategy sacrifices an higher number of injections per minute, allowed by the use of multiple injections in the same experiment, for more accurate and representative results. A Fault Injection Campaign can be defined as the execution of several Fault Injection Experiments.

After this obligatory concept definition we will describe Fault Injection in further detail. There are a multitude of Fault Injection techniques, nonetheless all of them can be classified according to a set of parameters:

- **Controllability**: Ability of controlling the injection of faults, both in time and space.
- **Observability**: Ability of observing the manifestations of the injected faults.
- **Repeatability**: Being capable of repeating an experiment and obtaining the same result.
- **Reproducibility**: Being able to reproduce the results of a fault injection campaign.
- **Representativeness**: How accurately can the emulated system, the Workload and the Fautload represent the real system.

The most common Fault Injection techniques are Hardware-Implemented Fault Injection and Software-Implemented Fault Injection. These techniques vary in the method used to inject the faults. While Hardware-Implemented Fault Injection makes use of specialized hardware components, Software-Implemented Fault Injection uses software for that purpose.

2.3.1 Fault Types

A fault can be classified, according to its dimension, either as an Hardware Fault, a malfunction of an hardware component, or a Software Fault, a malfunction due to a software defect. Typical software faults, injected by Fault Injection tools, are incorrectly assigned variables, wrong loop counter initialization, off-by-one overflows and incorrect comparison rules (e.g., use of <instead of <=) [30][10].

Hardware faults can be of various kinds, such as stuck-to-one, stuck-to-zero and bit-flip, and can affect a wide range of computer components, from CPU registers to memory transistors or communication buses.

Faults can also be classified, according to their persistence, as Permanent, Intermittent or Transient. A Permanent fault will remain active until the affected component is repaired. An Intermittent fault will appear and reappear in an apparently random fashion. A Transient fault will disappear after some time. This kind of fault can be caused by environmental effects, such as cosmic rays.

The emulation of a permanent hardware fault is considerably more elaborate than emulating a transient fault, since a permanent fault requires a manipulation every time the target hardware component (e.g., CPU register) is read, while a transient fault needs only 1 manipulation.

Previous research [26] has shown that the vast majority of hardware faults are transient.

Faults can also manifest themselves in a number of different ways. Some faults will cause the system or program to completely crash while others will cause it to enter an hang state. Some can have a less destructive effect, such as slowing down the performance of the system or making the system produce wrong or invalid output.

2.3.2 Software Implemented Fault Injection of Hardware Faults

The subset of fault injection tools that we focused in this thesis were Software Implement Fault Injection of Hardware Faults.

There are 2 approaches for the emulation of hardware faults by software: run-time injection and pre run-time injection. In run-time injection, faults are injected while the target system is executing the workload. This incurs a non-negligible overhead at runtime. In pre run-time injection, faults are inserted by manipulating either the source code or the executable image of the workload, before it is executed. This approach performs a trade-off between runtime overhead and setup time.

Since the beginning of research in this area, some tools have been developed that proved to be a significant improvement over their ancestors. Some of these tools are FIAT[5], FERRARI[22], FINE[23], FTAPE[40] and Xception[8].

One of the first tools to emulate hardware faults trough software was FIAT, developed at the Carnegie Mellon University. This tool injected faults by corrupting the code or the data area in the memory of the target program during runtime. FIAT supported 3 fault models: set-a-byte, zero-a-byte and two-bit compensation. Two-bit compensation consists of complementing any 2 bits of a 32-bit word. Single bitflips were not implemented due to the use of parity mechanisms in the memory.

FERRARI and FINE used more advanced techniques, with support for both permanent and transient hardware faults. FERRARI was capable of emulating address line, data line and condition code faults, by using the Linux function *ptrace*. While FINE focused on faults in the CPU registers, memory and memory bus.

FTAPE was designed with purpose of being used in benchmarking of fault tolerant commercial systems. FTAPE emulates single and multiple bitflips, as well as, zero and set faults, on the CPU, memory and disk components.

Finally, Xception, developed at *Universidade de Coimbra*, uses advanced debugging features present in recent processors. The use of these advanced features brings improvements over previous methods. It reduces interference with the workload, allows the recording of detailed information right before the injection and the use of complicated triggering conditions. Xception is capable of using stuck-at-one, stuck-at-zero and bitflip fault models.

2.3.3 Contemporary Frameworks

In this section, we describe the research for suitable contemporary tools to support the work of this

Analyzing the 3 propositions that stood out: KEDR, Linux Kernel's Fault Injection Framework and GDB Remote Debugging. However, unfortunately concluding that none of these 3 were fully adequate for our goals.

KEDR [24] is a framework devised for the analysis of kernel modules. Given that emphasis, the functionalities of this framework only apply to kernel modules and not to processes. This fact proves contrary to our needs. After selecting which module to inspect, the framework allows the interception of function calls, detection of memory leaks, simulation of resource usage and other uncommon events. It also features a so-called "Fault Simulation Facility", which simulates the results of a theoretical fault in the system. In conclusion, not only is KEDR unable to inject faults, it cannot target userspace process, and therefore does not fulfill our goals.

Since around kernel version 2.16.20, the Linux Kernel has featured a "Fault Injection Framework" [19], designed with the kernel and module developers in mind, so that uncommon code paths could be tested. This feature is not usually enabled in most Linux distributions and therefore requires a kernel recompilation before it can be used. The Fault Injection Framework is controlled through a file structure implemented over the RAM-based debugfs filesystem [14], which provides a simple interface to exchanging information between user-space and kernel-space. Some of the configuration options provided by this framework are:

- Type of fault
- Interval between faults
- Fault Injection probability
- Number of possible Fault Injections during process Lifetime
- Which process to target

This framework ticks some of the requirements for our Fault Injection Tool, however there are 2 critical points where it falls short. First of all, despite its name, this framework does not inject faults per se, but rather failures, by blocking certain kernel functions from operating. The other problem is that this framework requires the recompilation of the kernel, which increases the burden on the benchmark user.

GDB allows the remote debugging of a Linux kernel running in another system, through the use of a TCP or serial port connection. With this feature it is possible to inject breakpoints in the execution flow of the kernel, perform the fault injection and resume operation [32]. This method has its advantages: high controllability of injection and low intrusiveness in the target system. However it is considerably more complex than other options, is not as simple to automate and will also require kernel recompilation to enable debug symbols.

2.4 Virtualization

Virtualization is conceptually very similar to emulation, but with a key difference. Whereas in emulation a system pretends to be another system, in virtualization a system pretends to be two or more instances of the same system. As a matter of fact, nowadays Operating Systems make plenty of use of virtualization during their functioning. For example, when one process is running the Operating System virtualizes the CPU and Memory in a way which allows the process to act as if it was the only one on the entire system. If that process decides to use all of the CPU, the Operating System will comply with it, but will internally preempt other processes, so that every one has a fair share of the computing time. The same is true with Memory, trough the creation of a Virtual Address Space for each process. As should be clear by now, one key feature of virtualization is isolation. The same holds true for the virtualization used in cloud systems.

2.4.1 Hypervisor

In Cloud Systems and others, the virtualization capabilities which enable the existence of multiple Virtual Machines running on top of the same physical entity are provided by an Hypervisor. An Hypervisor is a software program that enables the execution of Virtual Machines on top of the same hardware.

Hypervisors can be classified in two groups, Native and Hosted, according to how close to the hardware they operate. A Native Hypervisor sits between the hardware and the Operating Systems of the Virtual Machines. Given their direct contact with the hardware these kind of hypervisors are usually the quickest. Examples of Native Hypervisors are KVM, Xen, Oracle VMServer and Microsoft Hyper-V.

Whereas a Hosted Hypervisor sits on top of one Operating System. Due to the overhead imposed by an extra layer, these hypervisors are at a disadvantage. Examples of Hosted Hypervisors are VirtualVox and VMware Workstation.

In Figure 2, the differences between a Native and a Hosted Hypervisor are displayed.

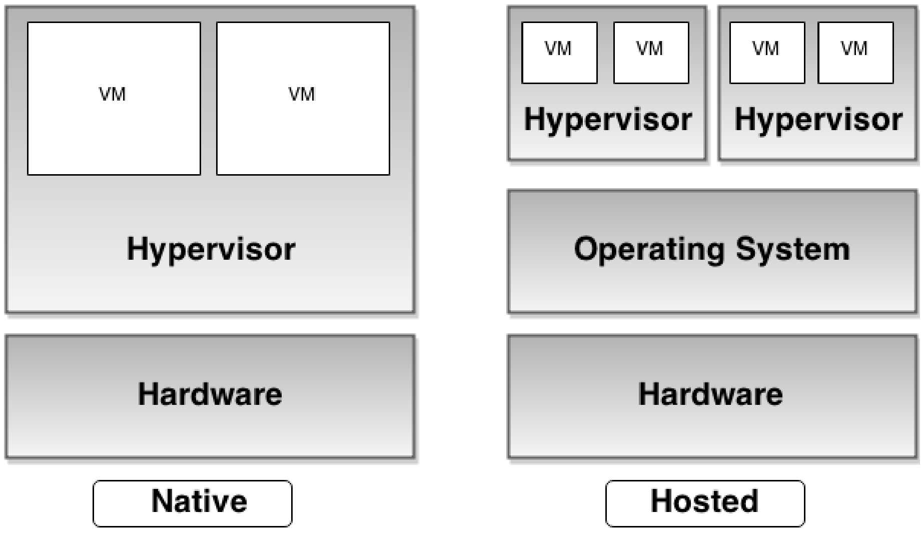

Figure 2: Differences between Native and Hosted Hypervisors

2.4.2 Virtualization Modes

In this thesis the chosen hypervisor was Xen, a well-known native hypervisor, and used in an array of cloud computing deployments, such as "Alibaba, Amazon Web Services, IBM Softlayer, Rackspace and Oracle" [28]. Xen is capable of deploying Virtual Machines through two different virtualization modes: Paravirtualization and Hardware-Assisted Virtualization [9]. In the Paravirtualization (PV) approach, Xen does not provide an environment exactly like the real one, but rather a very similar one, without functionalities that would be difficult to implement. This allows this mode to be used in hardware without virtualization support, at the expense of requiring a slightly modified guest kernel. While modifying a Linux or any other open-source kernel for supporting Xen Paravirtualization as a Guest is not very complicated, the same cannot be said of an Operating System such as Windows, where the kernel code is not public. For this reason it is impossible to run Windows by Paravirtualization.

The most important difference that the guest Operating Systems have to face when being virtualized trough Paravirtualization is the fact that instead of running in the usual Ring 0, they are forced to run in Ring 1, due to the occupation of Ring 0 by Xen's hypervisor. Now in Ring 1, the Operating System sees itself deprived from the capabilities to perform privileged accesses to the hardware. To solve this problem Xen uses the so called, hypercalls. Hypercalls are an idea not unlike syscalls, where the guest Operating System can communicate with the hypervisor by passing values and then calling an interrupt.

With the advent and popularization of hardware support for virtualization, a new virtualization mode has become available, Hardware-Assisted Virtualization (HVM). This virtualization mode does not require the use of a custom guest Operating System, because it uses hardware capabilities that make it possible to run the guests in the precise same environment as they would in a dedicated system. Most of these hardware virtualization technologies focus on creating new rings in the CPU, so that both Xen and the guest Operating Systems can run in a privileged state.

In Figure 3, the privilege rings for Bare-Metal, Paravirtualization and Hardware-Assisted Virtualization are shown.

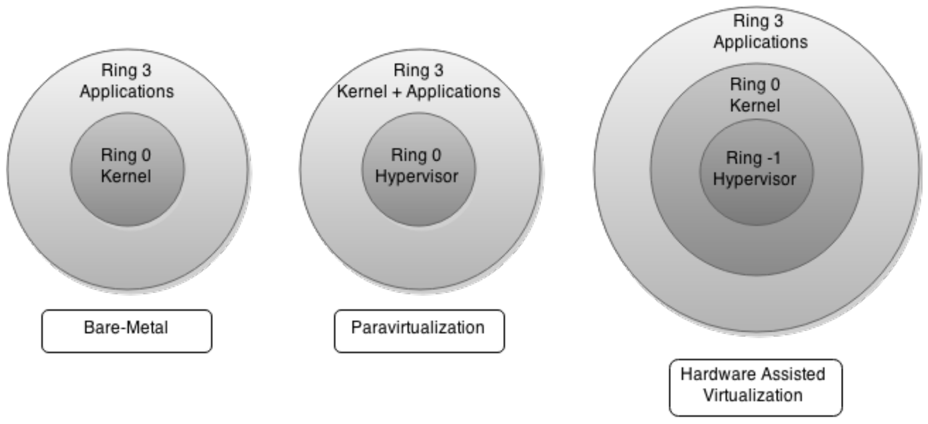

Figure 3: Privilege rings for Bare-Metal, Paravirtualization and Hardware-Assisted Virtualization

3 Fault Injection Tools

What eventually became this thesis main focus of attention and biggest contributions were the fault injection tools capable of targeting different components of a virtualization system. In total 3 different fault injection tools were created, which one specially designed to target one area of the virtualized system:

- Kernel module to inject register and memory bitflips in applications
- Xen hypercall to inject memory bitflips inside Xen's protected memory
- Xen assembly code modification to inject register bitflips inside executable code (e.g., hypercalls)

This section provides an in-depth overview of each tool.

3.1 Injecting in a process

We will describe the implementation, limitations and motivation behind this tool, which was the first one to be created, still during the first semester.

3.1.1 Motivation

The first fault injection tool was built in order to support the experiments inside the guest virtual machines. This tool is capable of performing both register and memory bitflips, in userspace and kernelspace processes. It was implemented as two Linux kernel modules that when loaded will receive the necessary parameters and perform the injection. If a register bitflip is chosen, the tool will modify the registers of a process that the kernel saves in memory. In the context switch to the targeted process, the kernel will load the corrupted register and proceed with the execution flow. This allows the tool to precisely limit the impact of the bitflip solely to one process. If a memory injection is chosen, the tool will obtain the memory pages of the process, choose one of them randomly, load that page into the module's own memory address and perform a bitflip in a random position.

3.1.2 Implementation

In order to improve code modularity we implemented the tool as two different and independent modules, that share the same code base. The first module is implemented in file registers.c and includes the following functions:

- void save_injec()
- void register_fault()
- int __init mod_init()
- void __exit mod_cleanup()

The *mod_init* and *mod_cleanup* functions are obligatory for any kernel module and will be called by the kernel when loading and removing a module. In our case *mod_cleanup* is left empty, because there are no tasks that need to be performed. The function *mod_init* is the main function of each model, and contains the module's logic and sanity checks. Function *register_fault* encompasses the act of performing the bitflip in a register. It starts by disabling preemption in the kernel, this enforces that no other process can interrupt this function execution, avoiding hard to trace problems. Then the structure inside the kernel that holds the process we want to target is found and the register is corrupted. Before restoring preemption to the kernel, we call function *save_injec*. This function will save various information about the injection to stable storage, so it can be analyzed later. Stable storage is a storage that guarantees that any write operation is atomic, this means that if we read the written-to portion of the disk after a write, we must either get the written data or the data that was there before the write operation.

The second module is implemented in memory.c:

- void save_injec()

- void corrupt()
- unsigned int size()
- void memory_fault()
- int __init mod_init()
- void __exit mod_cleanup()

As happened in registers.c, *mod_cleanup* is left empty. *mod_init* begins by verifying if the parameters were correctly passed and to what memory flags we should limit the injection (Read-only, write-only or executable memory). Function *memory_fault* holds the code that performs the injection. It starts by disabling kernel preemption, locating the kernel structure of the process, and obtaining all the memory pages belonging to the process that obey our flag choice. After this step, the function will choose a random page and byte to bitflip. Then it will need to map the page into memory, since it belongs to another process and can have been swapped to disk. This will load the page from its location (e.g., swap, RAM) into the active memory area of our module. Performing the bitflip is now a easy task that is implemented in function *corrupt*. After the bit is flipped the page is unmapped and the information about the injection is saved to stable storage (*save_injec*). Finally kernel preemption is re-enabled.

Functions that can be shared among both modules are kept in file utility.h:

- unsigned int get_random_int_compab()
- int readProgramCode();
- struct page *walk_page_table();
- int getRIP();
- struct file* file_open();
- int file_write();
- int file_read();
- void file_close();

get_random_int_compab was implemented as a compatibility function for some kernel versions that did not possess *get_random_int* function. It uses the kernel's *get_random_bytes* to return a random unsigned integer value. *readProgramCode* returns the content of the memory area defined in the parameters. *getRIP* is used to read the memory content in the nearby areas to where the process Instruction Pointer register is, this information can be useful for posterior analysis, by extracting the instructions being executed at the moment of injection, it makes use of the *readProgramCode* function. *file_open*, *file_write*, *file_read* and *file_close* are functions used to aid the filesystem operations that the modules need to perform. The code of these functions took strong influence from [16] .

One crucial aspect of fault injection is defining the moment of injection, or the trigger. To provide this functionality, the tool resorts to a userspace program that provides temporal triggering, this means, triggering after a certain period of time has elapsed. This program, located in timer.c, receives as parameters the amount of time (in ms) to sleep, the name of the processes to target and the command line to run (usually an *insmod* that loads the module into the kernel). At the beginning, the program will sleep for the desired amount, using the high-resolution *nanosleep* function. After waking up, the program will use the filesystem structure existent in */proc* to obtain the PID of every process that we are looking for (e.g., every process of Apache). Finally, the program will randomly choose one of the processes and execute the requested command, but before it will elevate its privileges to root, by using *setuid*.

Another important feature of this tool is the logging of execution information to stable storage, for posterior extraction and analysis. The information we chose to store in the target host consists of:

- PID of target process
- Value of the register before injection
- Value of the register after injection
- Value of the Instruction Pointer Register
- Program opcodes in the near region of the current execution flow

At the same time, the host who launches and commands the campaign stores more information, namely:

- Type of Fault Model (Register or Memory bitflip)
- Timestamp of start of the tool
- Interval (in ms) before the fault is injected
- Register or position in memory targeted
- Bit chosen to be flipped

All this information is crucial to completely understand when, how and why one fault injection lead to a particular outcome.

In Figure 4 a rudimentary diagram of the usual flow when using this tool is displayed.

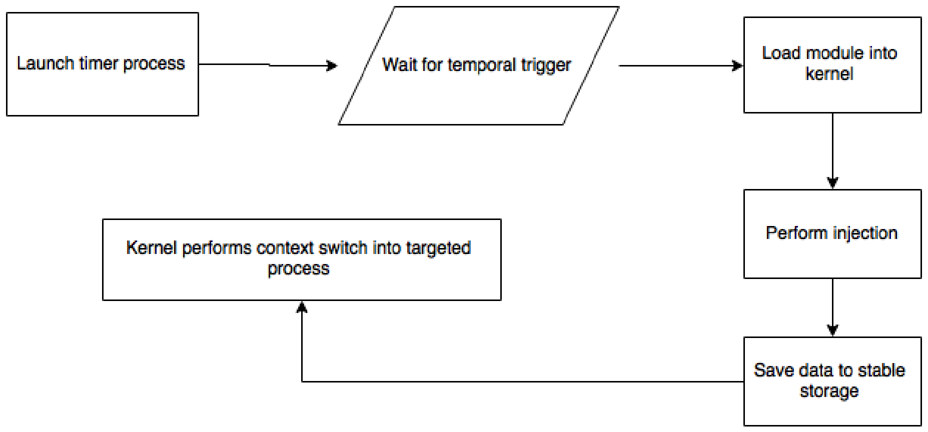
Launch timer process
Wait for temporal trigger
Load module into kernel
Perform injection
Save data to stable storage
Kernel performs context switch into targeted process

Figure 4: Sequence flow of the usage of the fault injector

3.1.3 Limitations

The choice of using a kernel module limits the scope of this tool to Linux systems using recent kernel versions (from Linux 3.x up, but older versions can be supported with small changes to the tool). Yet the impact of this disadvantage is reduced given the proliferation of Linux in the Cloud Computing field [12] [34]. The hardware architecture is also limited to x86-64, the same of our testing setup, but can easily be ported to other architectures with minimal code modifcations.

Whereas integrating the triggering mechanism into the kernel module would be the preferred method, due to the higher precision it would provide versus a userspace mechanism, this was proven unfeasible due to inherent limitations of performing disk I/O while in a interrupt context. [29]. Nonetheless, the userspace program makes an effort to reduce its overhead as much as possible.

3.2 Injecting in Xen's memory

The motivation, implementation and limitations of this tool are described in the following subsections.

3.2.1 Motivation

By default Xen protects its own memory from access by lesser privileged layers, such as dom0 and the guest virtual machines. [44, 38] This improves the security of the system but presents an obstacle to our objectives. In order to work around this limitation the supervisors suggested to modify Xen's source code and add a way for a lower privilege domain to request a memory bitflips.

Our approach was chosen after a review of the literature, where previous options were highly complex and hardware dependent, and consists in modifying the source code by adding a new hypercall. This hypercall performs a bitflip in Xen's memory and receives two parameters, the memory position and bit to flip. As with any other hypercall it can be called from the dom0 or the guest machines. The disadvantage of this method is the inherent modification of the system, which can cause unaccounted differences versus the original system, and slightly modify the results. As to reduce this effect we strived to keep the modifications to a minimum.

3.2.2 Implementation

The hypercall code is presented below:

```c
long do_bitflip(unsigned long input, int pos)
{
    unsigned long  * t;
    t = ((unsigned long *) (0xffff82d080000000+input));
    (*t) ^= 1 << pos;
    return 1;
}
```

The hypercall's code is pretty straightforward, but it should be explained that the value 0xffff82d080000000 is added to the input because it is the location where the memory assigned for Xen's use starts, as defined in one of Xen's configuration file. The code of the userspace tool that is used to call the hypercall is:

```c
int main(int argc, char **argv)
{
    xc_interface *xch;
    int ret;
    unsigned long input_number;
    int pos;

    xch = xc_interface_open(0,0,0);
    if (!xch)
        errx(1, "failed to open control interface");

    input_number = atoi(argv[1]);
    pos = atoi(argv[2]);

    ret = do_bitflip_hypercall(xch, input_number, pos);

    if (ret == 0)
    {
        errx(1, "ret == 0\n");
        fflush(stdout);
    }

    xc_interface_close(xch);

    return ret;
}
```

3.2.3 Limitations

Due to the low-level nature of this tool it would dramatically increase its overhead if we added a logging functionality that would allow us to store information about each injection. For this reason this functionality has not been implemented.

3.3 Injecting in Xen's registers

The motivation, implementation and limitations of this tool are described in the following subsections.

3.3.1 Motivation

Up until this point the last major functionality that we missed was the injection of faults in registers during the execution of Xen's code. In order to fulfill this objective the supervisors challenged me to evaluate a different approach from previous tools. One that would use compile-time (pre run-time) injection instead of run-time, by modifying the assembly code of the most called Xen hypercall and adding a small set of assembly instructions in between the original instructions. After reflection, I deemed this idea to have a big possibility of success and we set off developing it.

3.3.2 Implementation

After analyzing which hypercall we would target, we developed the payload that would be used to modify the registers. In order to reduce the overhead and intrusiveness of our approach we aimed at using the lowest amount of instructions required to attain our goal. Given the limitations of assembly instructions we developed 2 different payloads: one that would work with most common-purpose registers (AX, BX, CX, ...) and another specifically designed for the Instruction Pointer register.

Below we can see a sample of both payloads. Please take notice that the intermediate registers can and must change according to the register being targeted (e.g., we cannot use AX as a intermediate register when we are trying to perform a bitflip in AX).

```
    .globl  contador
```

```
 2      .bss
 3      .type   contador, @object
 4      .size   contador
 5      contador:
 6      .zero   1
```

```
 1      pushq %rax
 2      cmp $1, contador(%rip)
 3      je final
 4      pushq %rdx
 5      rdtsc
 6      salq $32, %rdx,
 7      or %rax, %rdx
 8      movabsq $valor, %rax
 9      cmp %rdx, %rax
10      popq %rdx
11      jg final
12      movabsq $mascara, %rax
13      movb $1, contador(%rip)
14      xorq %rax, %registo
15      final:
16      popq %rax
```

At first we add the declaration of a new variable to the header of the assembly file. This variable (in this case named *contador*) will only use 1 byte, and serves as a boolean that indicates whether a bitflip has already been performed.

The first line pushes the first intermediate register into the stack (in this case we are using *rax*), so that its original value is not tainted. In our code we will need the use of 2 intermediate registers (e.g., *rax* and *rdx*). The next line verifies if a bitflip has already been done (contador == 1), if this is true we will not need to continue any further (je final). The fourth line pushes the second intermediate register into the stack. In the fifth line we call the *rdtsc* (Read Time Stamp Counter) instruction. This instruction returns the amount of CPU cycles since the start of the system. This 64-bit value is by default saved to the rax and rdx registers. Line 6 and 7 are used to copy the 64-bit value into just 1 register, so that we can perform the comparison. Line 6 performs a arithmetic shift left by 32 bits. Line 7 performs a OR between rax and rdx, and saves the result into rdx. In line 8, *valor* is loaded into rax. *valor* is a placeholder for the numeric value of CPU cycles that we want to wait before performing the bitflip. Line 9 performs the comparison between these two values, held in rax and rdx and sets the flags accordingly. Next line performs a jump to a label near the end of our payload when the amount of cycles is not yet enough to perform the injection. And we take this chance to pop one register that will be no longer needed from the stack. If there are enough cycles we continue to line 12 where we move *mascara* into register rax. *mascara* is another placeholder, where the

XOR mask is kept. Given that we are dealing with 64-bit values, a special version of the *mov* instruction (*movabsq*) had to be used. Next line we set the the *contador* variable to 1, meaning that the bitflip has already taken place. Finally, the next line will perform the bitflip itself, by xoring our target register (placeholder *registo*) with the mask loaded into rax. We end by popping the last register from the stack. From now on the code will continue its execution as usual, apart from the bitflip.

The second payload is similar to the previous one but uses a different method to read and modify the IP register. By default the previous method of directly assessing the register (xorq %rax, %registo) is not supported for the IP register. For this reason we used the long jump instruction to modify the register and the Load Effective Address (LEA) instruction to obtain the current value of the IP.

At first we needed to add a new variable to the file's header, to be used when holding the Instruction Pointer value before the jump.

```
    .globl  contador
    .bss
    .type   contador, @object
    .size   contador, 1
    contador:
    .zero   1
    .globl  val_reg
    .bss
    .type   val_reg, @object
    .size   val_reg, 8
    val_reg:
    zero   8
```

```
    pushq %rax
    cmp $1, contador(%rip)
    je final
    pushq %rdx
    rdtsc
    tsalq $32, %rdx
    or %rax, %rdx
    movabsq $valor, %rax
    tcmp %rdx, %rax
    jg final
    movabsq $mascara, %rdx
    movb $1, contador(%rip)
    lea (%rip), %rax
    xorq %rdx, %rax
    movq %rax, val_reg(%rip)
    popq %rdx
    popq %rax
    ljmp val_reg(%rip)
    final:
    popq %rdx,
    popq %rax
```

3.3.3 Limitations

As happened with the previous tool, the very low-level nature of this fault injection tool lead us not to implement any logging functionality.

4 Mechanism for Recovery of Virtualization Infrastructures

The needs that arose during this thesis lead us to the development of an external watchdog timer that is capable of assessing when the virtualization infrastructure is displaying incorrect behaviour to the clients, and also capable of restoring the correct service even if the infrastructure is completely unresponsive. In this section we present an detailed overview of this mechanism, a brief introduction to the technology that supports it and a benchmark of the performance of the watchdog.

4.1 Watchdog Overview

During the development of our experiments we were faced with the need to infer when the system was unresponsive and to be able to restart it in a automatic fashion, even if the hypervisor and the VMs are completely inaccessible. I researched this question and developed a solution that makes use of an Intel technology widely deployed in virtualization servers and other workspace computers: Intel AMT. Intel AMT consists of an imbued CPU inside the main CPU, which runs completely parallel and decoupled from the main CPU, and by listening on the network card is capable of providing many management operations over the network independently of the state of the Operating System (e.g., hanged). In particular, for our watchdog implementation, we use AMT's capability of restarting the computer at our request.

The watchdog needs to be installed in an external computer that is connected to the same network of the systems under watch, and operates by periodically performing responsiveness checks. These responsiveness checks can be defined by the network administrator, but will usually consist of a simple ping check followed by an ssh attempt or an attempt to simulate a client of the services (e.g., visitor in webpage). After a predefined timeout, if there is still no answer, the watchdog will force the server to restart by sending a specially crafted AMT packet. Finally, the watchdog waits for the server to restart and then reloads the Virtual Machines by executing a script installed in the server.

In Figure 5 a simple graphical overview of this system is presented.

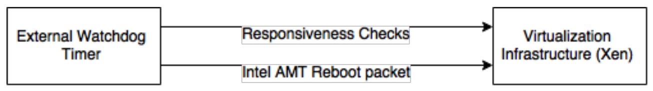

Figure 5: Overview of watchdog system

4.2 Overview of Intel AMT

Intel AMT (Active Management Technology) [20] is a technology embedded in some Intel-based platforms that improves the ability of organizations to manage enterprise computing facilities. It operates independently of the processor or operating system, and therefore remote applications can access Intel AMT even when the system is turned off or hanged, as long as the system is connected to a power line and to the network.

Some often advertised use cases for Intel AMT are:

- Discover all of your computing assets: Intel AMT stores information about the hardware and software in non-volatile memory. This feature allows IT management to discover hardware and software assets even when the PCs are powered off.

- Heal systems remotely regardless of system state: Intel AMT provide out-of-band access and management capabilities that allow IT management to fix system problems even after OS failures. Intel AMT also supports alerts and logs to help the timely detection of problems.

- Protect against malicious software attacks: Intel AMT helps an organization to protect it's network by making it easy to keep software and anti-virus products up-to-date. Third party software can store version numbers or policy data in the non-volatile memory for off-hours updates.

- Contain the effect of malware and platform misuse: Intel AMT is able to confine virus infections by sealing the infected network elements from the rest of the network. It can detect if anti-virus software is executing, and send a report in case it is not.

There are 3 methods that a remote application can use to communicate with Intel AMT: Simple Object Access Protocol (SOAP) Messages, Proprietary Redirection Protocol and WS-Management. SOAP is a well-known XML-based protocol for exchanging information in a distributed and decoupled way. It is composed of 3 parts:

- An envelope that states what are the message's content and how to process them
- Encoding rules for expressing the instances of the application-defined data types
- Convention for remote procedure calls

Proprietary Redirection Protocol is a Intel proprietary protocol that can be used to access Intel AMT's features. WS-Management (”Web services for management”) is a recent DMTF (Distributed Management Task Force) standard for managing devices in a network using an object-oriented approach. It should be noted that WS-Management is a layer over SOAP messages.

4.3 Benchmark Results

In order to validate and benchmark the performance and effectiveness of our watchdog a simple experiment was performed where the system was purposely forced into an unresponsive state and the watchdog was forced to act. As to force the machine into an unresponsive state we shortened the duration of the experiment to 2 minutes and injected register bitflips only in IP, BX and SP registers, on kernelspace processes in dom0. The time taken to restart the system and the success of the recovery action were measured and are presented in Table 2.

System state		Hypervisor recovery			VM recovery		
Total experiments	206	Min	Max	Avg	Min	Max	Avg
Hang	203	30 s	34 s	31.9 s	54 s	103 s	75.2 s
No effect	3						

Table 2: Evaluation of the recovery implementation.

As can be seen, despite the relative simplicity of our approach, both the success and time overhead had very good results. In our case, the system did not take longer than 137 seconds to recover (maximum of hypervisor and VM recovery), with most of the time being spent in the reloading of the VMs, which will take a different amount of time according to the amount and size of the VMs in each scenario.

4.4 Conclusion

This section presented and benchmarked the watchdog mechanism which was developed by the studend during the course of the thesis. Despite its success in performing the role it was intended for, it should be noted that there are a few limitations and disadvantages with the present implementation of the watchdog timer, namely:

- The watchdog timer needs to be running in an external component that is connected to the same network of the virtualization infrastructure. First of all this will bring an additional hardware cost overhead that can be non-negligible. Furthermore a failure in the connection between the watchdog timer and the remaining system will completely disable the watchdog operation. Nonetheless there are plenty of well-known technical solutions to this limitation, such as using two or more redundant connections, but the implementation of such feature was left out of our scope.

- User-defined responsiveness checks should detect all cases where system needs to be restarted, a failure to do so can create cases where the watchdog timer fails to restart the system because it is unable to correctly comprehend that such action is needed.

5 Experimental Results

With the intent of characterizing the behaviour of a virtualized application under the influence of soft errors and of putting our recently developed tools to test, we conducted a series of fault injection campaigns. The campaigns and their results are presented and analyzed in this section.

5.1 Definition

Across this thesis several different campaigns were undertaken, which despite having the same goal in mind, took different approaches to reach it. These campaigns were:

- Injection in Apache's processes inside VM
- Injection in OS's processes inside VM
- Injection in Xen's processes in dom0
- Memory bitflips in Xen's memory
- Register bitflips in Xen's hypercalls

Despite the differences between them, namely in what component the faults were injected, there are many properties that are kept the same. All the campaigns used two different guest VMs running the same system image and the same workload, VM1 or VM2, also known in some campagins as Fault VM and Control VM, respectively. This difference in denomination is due to the fact that while in the campaigns that do not inject inside privileged VMs, there is absolutely no difference between the two VMs, when we injected inside a privileged VM, we chose VM1 (Fault VM) for that effect, hence the name. Both VMs were executed side-by-side.

The Faultload is a transient hardware fault, in the form of a bitflip in one of the available registers or memory. Both the bit to flip, the register or memory location to target are chosen in a random way, using an uniform distribution PRNG.

The Workload consists of a scenario where multiple external HTTP clients, simulated with use of one external computer (Control & Command), access one page in an Apache webserver running in both VMs. This webserver serves only one page, where its content is the result from an SHA1 hash, using as input a stream obtained from /dev/zero. A similar approach was taken in "Challenges and Opportunities with Fault Injection in Virtualized Systems" [27]. This workload stresses particularly the system's CPU, and, to an extent, its memory. The size of the input stream can be defined by passing a HTTP GET parameter, allowing a simple and quick configuration. In our campaigns the size was of 1024Mb. This relatively large size was chosen so that the Transactions per Second could be relatively small, and therefore the failure of one process due to a fault could have more impact and be easily detect than when the input size is small and the process' life is short.

In order to simulate the HTTP clients, JMeter was used, configured to use 10 threads, with a 30 second ramp-up, and a 600ms with 100ms deviation pause between requests.

In order to save the disk state between executions, a LVM snapshot was created of each VMs' logical volume. A LVM snapshot is nothing more than another logical partition which saves the modifications done to the original volume from there on. When a run has finished, these modifications are of no use to us, and therefore we can drop this snapshot and start again from the initial state. The Xen snapshot feature was also used to perform a similar action with the memory state of both VMs. Xen is capable of saving a VM state at any given point in time to a file, and later, to restore that state back. With such capability it was possible to reload a "clean" state into execution, each time that the run had finished and another run was about to start. Without these two features, the downtime in-between executions would have significantly increased, resulting in a decrease in number of runs per unit of time.

The metrics obtained can be grouped in two categories: performance and isolation metrics. The performance metrics are Response Time (in ms), Transactions Per Second and Network Performance (both uplink and downlink). These metrics are calculated by the software that simulates the clients (JMeter), and therefore does not in any way interfere with the VMs' execution. The isolation metrics refer to the correctness of the output provided by the webserver, for both the Target VM and the Control VM. These metrics are also obtained by JMeter, simply by analyzing the provided HTTP response against a predefined expected response. These metrics allowed us to understand what effect the fault had in the Fault VM and its neighbouring VM, the Control VM.

In order to widen the scope of information obtained from the VMs, a simple and lightweight probe is launched in both VMs. This probe records data regarding the system's CPU, Memory and Swap usage along the experiment run, and saves it in a CSV file. This CSV file is later extracted when the experiment has finished.

The state of the hypervisor at the end of an experiment is assessed by the use of a script that performs a ping check and attempts an SSH connection, followed by the execution of some commands whose output can be used to verify the health of the system. With this information the hypervisor state is classified either as Responsive or Unresponsive.

Both Virtualization Modes were used in the campaigns and the results specify explicitly which mode was used.

The experiment flow and stages for the experiment campaigns that targeted solely processes running in the guest VMs (FaultVM) can be previewed in Figure 6.

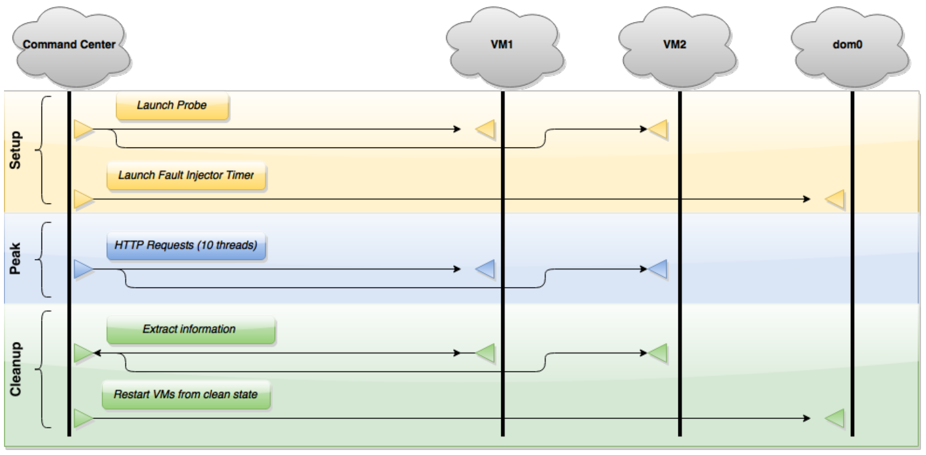
Command Center
VM1
VM2
dom0
Setup
Launch Probe
Launch Fault Injector Timer
Peak
HTTP Requests (10 threads)
Cleanup
Extract information
Restart VMs from clean state

Figure 6: Sequence of the campaigns that inject inside guest VM

Each experiment run of this kind had the duration of sensibly 420 seconds (or 7 minutes), 330s (or 5 and a half minutes) of which 220s correspond to the time where the workload is being executed, and the remaining 110s are divided among the time of setup before the start of each run and the cleanup time at the end. The fault is injected in one process running inside FaultVM in an interval between [30s; 4m]. This interval guarantees that the workload had time to warmup and is now working at peak throughput, and also provides enough time before the end of the experiment for the fault to be propagated.

Meanwhile, the experiments that targeted the privileged domain (dom0) had a slightly different flow as described in Figure 7. Instead of launching the fault injection timer in a guest VM, it is launched inside dom0.

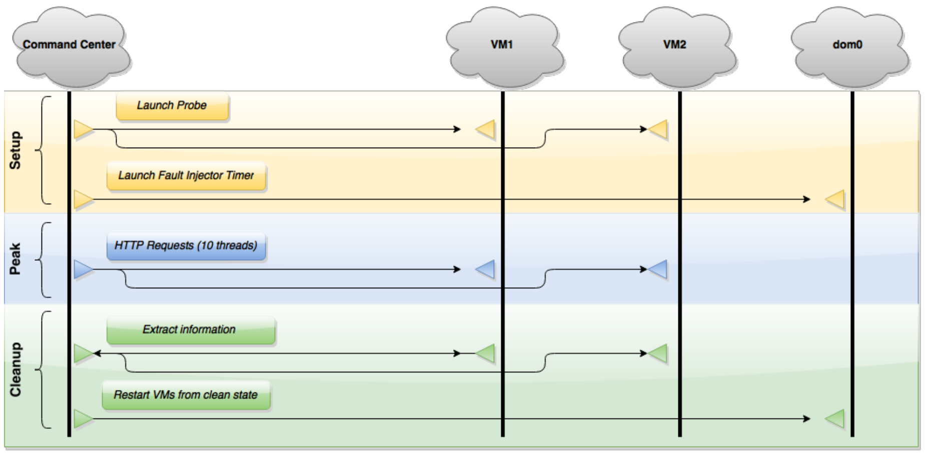
Command Center
VM1
VM2
dom0
Setup
Launch Probe
Launch Fault Injector Timer
Peak
HTTP Requests (10 threads)
Cleanup
Extract information
Restart VMs from clean state

Figure 7: Sequence of the campaigns that inject inside dom0

The campaign that consisted in injecting memory bitflips had the flow presented in Figure 8, where we can see that the injection is triggered by a userspace tool located in dom0, which interacts with Xen through an hypercall. The entire system is restarted through the use of the Intel AMT feature present in the CPU, in order to assure that a faulty hypervisor does not block the system from rebooting.

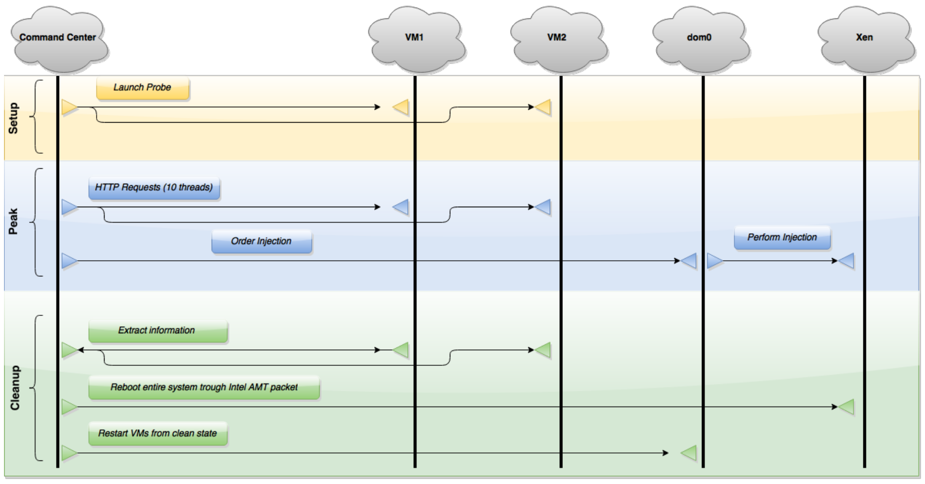

Figure 8: Sequence of the campaign that inject in Xen memory

The campaign that injected bitflips in registers during Xen hypercalls had the flow that appears in Figure 9. This flow is quite different from previous ones, because it consists of 4 phases instead of the usual 3. Other than the Setup, Peak and Cleanup phases, a new phase called "Pre-Setup" was added.

This phase, which happens before any other, is where we compile and install a version of Xen that has our register injection payload inside. Since we already injected the fault injection payload inside the code we do not need to load or call any userspace program.

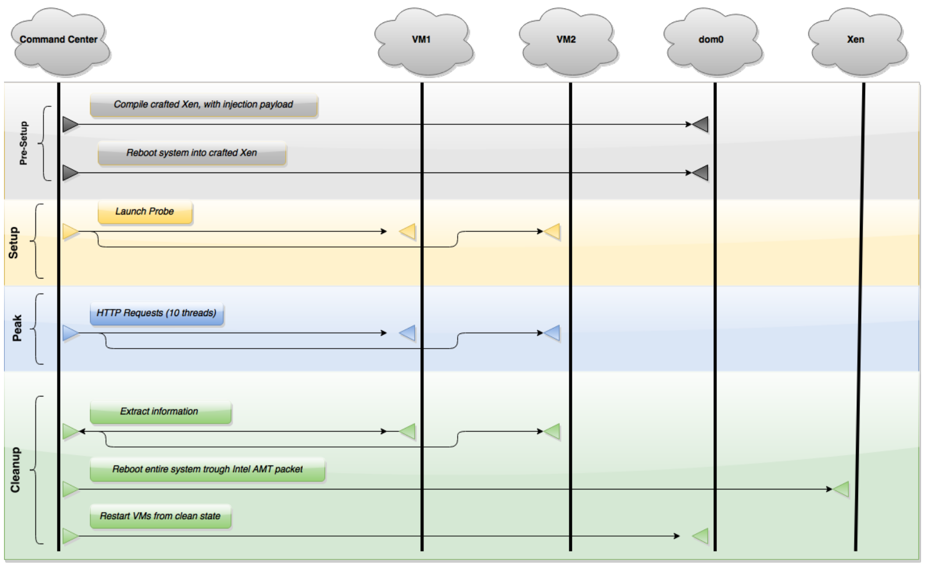
Command Center
VM1
VM2
dom0
Xen
Pre-Setup
Compile crafted Xen, with injection payload
Reboot system into crafted Xen
Setup
Launch Probe
Peak
HTTP Requests (10 threads)
Cleanup
Extract information
Reboot entire system trough Intel AMT packet
Restart VMs from clean state

Figure 9: Sequence of the campaign that injects in Xen registers

5.2 Physical Setup

In order to provide a physical setup for the experiment, a Fujitsu Celsius desktop computer was arranged.

The specifications of this computer were carefully chosen to match those used in Cloud Computing datacenters worldwide [1, 3, 21]. This ensured that the results from the campaigns closely represented the ones that would be obtained in the real-world.

In Table 3 a brief synopsis of the most important components of the system is provided.

CPU	Intel Core i7-4770, 4 physical cores running up to 3.9GHz
Cache	8 Mb
Virtualization Technologies	VT-x, VT-d, EPT
RAM	8Gb DDR3
HDD	1TB
SSD	120GB Samsung EVO II

Table 3: Hardware Specifications of Virtualization Server

In the same way that the hardware strives to reflect a real world system, so does the software. The chosen hypervisor was Xen version 4.4.1, released on September 2014. At the time of writing the latest stable Xen version was 4.5, released January 2015.

The dom0 operating system is CentOS 7.0 with kernel version 3.11.1. In order to support all the features that Xen can take advantage of, the kernel had to be recompiled using a configuration file with full Xen host support. The choice of CentOS for dom0 can be dismissed by some as not ideal given the representativeness of XenServer in the real-world. While the presence that XenServer has in deployments all over the world cannot be denied, it must be noted that XenServer is a package which, among other things, includes a control domain (dom0) based on CentOS [36]. Proving not to be much different from our setup.

The domU operating system is Debian 7.7, with kernel version 3.0.2. Debian was chosen because of its fame as a stable and reliable server-oriented operating system.

For the experiment's workload, Apache 2.2.22 was used, as provided by the Debian repositories, including the default configuration files. On the Control&Command system, JMeter 2.12 was used to emulate the client requests to the webserver.

5.3 Classification of Failures

In our experiments we classified the results according to a client oriented view, this means we classify the results as the client perceives them, or, to put it another way, according to what impact they have in the output sent to the outside of the virtualized system. It must be added that at the end of each fault injection run we assessed the responsiveness state of the hypervisor through the use of correctness tests (such as performing a ssh connection and obtaining the output of a few utilities). The behaviour can be classified in different groups:

- **Incorrect Content**: Occurs when the webserver returns syntactically correct HTML but with an incorrect hash inside. This failure mode is perceptible to the user but also the most serious and difficult to handle at system-level, because wrong data can be undetected unless it is checked against a redundant computation.

- **Corrupted Output**: In this case the socket remains open, however the data stream contains solely corrupted data (syntactically incorrect HTTP packet, invalid HTML code or just pure gibberish). For example, a browser would handle a similar occurrence by displaying an error message. Therefore, the client is capable of detect this behaviour, but since there is no data to interpret, this behaviour is not as nefarious as that of *Incorrect Content*.
- **Connection Reset**: The TCP connection between the server and the client is reset by the server's network stack, by sending a packet with RST flag set.
- **Client-Side Timeout**: One or more of the clients fails to receive a response inside the predefined time limit (20 seconds), and issues a client-side timeout. The time limit of 20 seconds was by us considered enough for HTTP communications, as an example, the keep-alive mechanism usually maintains a connection open for 5-15 seconds.
- **Hang**: The service stops producing output and answering any subsequent requests. Eventually all connected clients will issue a client-side timeout. In some cases, due to timing aspects outside of our control, some experiments are classified as client-side timeouts when the real behaviour should be Hang.
- **No Effect**: The injected error has no visible effect on the provided service, both in terms of performance (latency, throughput, ...) and correctness.

5.4 Results and Analysis

The results of the various campaigns are presented and analyzed here.

5.4.1 Injection in Apache's processes inside VM

At first we used the HVM virtualization mode and obtained the results presented in Table 4:

Virtual machine 1 (Fault VM)		Virtual machine 2 (Control VM)	
Incorrect content	4 (0.4%)	Incorrect content	0 (0%)
Corrupted output	2 (0.2%)	Corrupted output	0 (0%)
Connection reset	12 (1.2%)	Connection reset	0 (0%)
Client-side timeout	130 (12.7%)	Client-side timeout	0 (0%)
Hang	5 (0.5%)	Hang	0 (0%)
No effect	876 (85.3%)	No effect	1027 (100%)
Hypervisor responsive	1027 (100%)		

Table 4: Outcomes of fault injection targeting application processes within a virtual machine, in HVM mode

As can be seen by the 0% wrong output in the Control VM, it appears that faults injected in the FaultVM do not propagate to the ControlVM. This conclusion was further reinforced when we analyzed the performance of both virtual machines and verified that there was no performance difference between machines and between a previous baseline run. It is therefore safe to assert that the inter-VM separation provided by the hypervisor was kept intact.

In order to fully explore Xen's virtualization capabilities, we executed the same campaign but while using the PV virtualization mode. The results are presented in Table 5.

Virtual machine 1 (Fault VM)		Virtual machine 2 (Control VM)	
Incorrect content	6 (0.6%)	Incorrect content	0 (0%)
Corrupted output	1 (0.1%)	Corrupted output	0 (0%)
Connection reset	8 (0.8%)	Connection reset	0 (0%)
Client-side timeout	71 (7.3%)	Client-side timeout	0 (0%)
Hang	6 (0.6%)	Hang	0 (0%)
No effect	876 (90.5%)	No effect	968 (100%)
Hypervisor responsive	968 (100%)		

Table 5: Outcomes of fault injection targeting application processes within a virtual machine, in PV mode

Once again there was no breach between virtual machines. Equipped with the results for both virtualization modes, it is possible to compare them and notice the differences inherent to the method used to implement virtualization. A straightforward conclusion that can be taken from both runs is that PV mode presents a small but noticeable lower percentage of produced failures (9.5% vs 14.7%). The explanation for this fact must be linked to the differences on how each virtualization mode achieves their purpose. On the other hand, there is no big variation between the classification of the produced failures between virtualization modes, where Client-Side timeouts are the most common occurrence.

In Figure 10 and Figure 11 the distribution of failures modes according to registers is presented for both PV and HVM virtualization modes respectively.

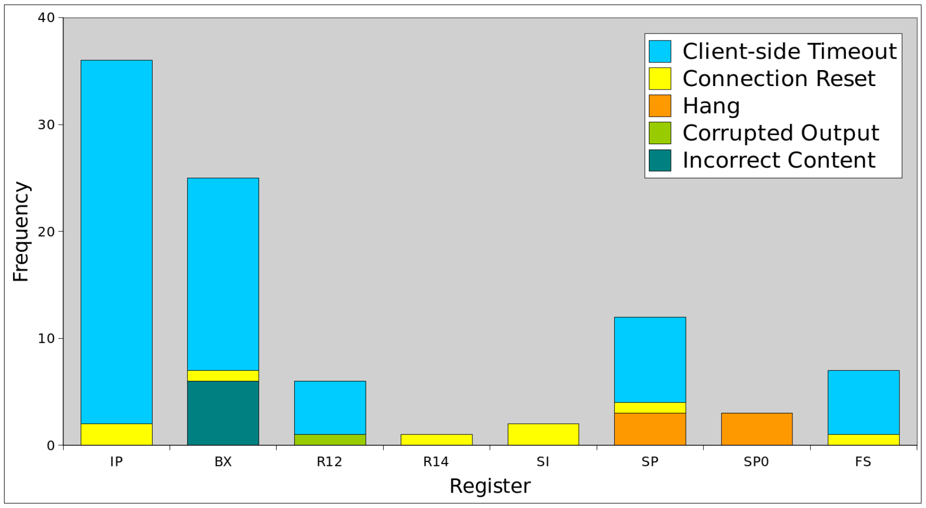

Figure 10: Distribution of failure modes across processor registers, for injections in application processes, in PV mode

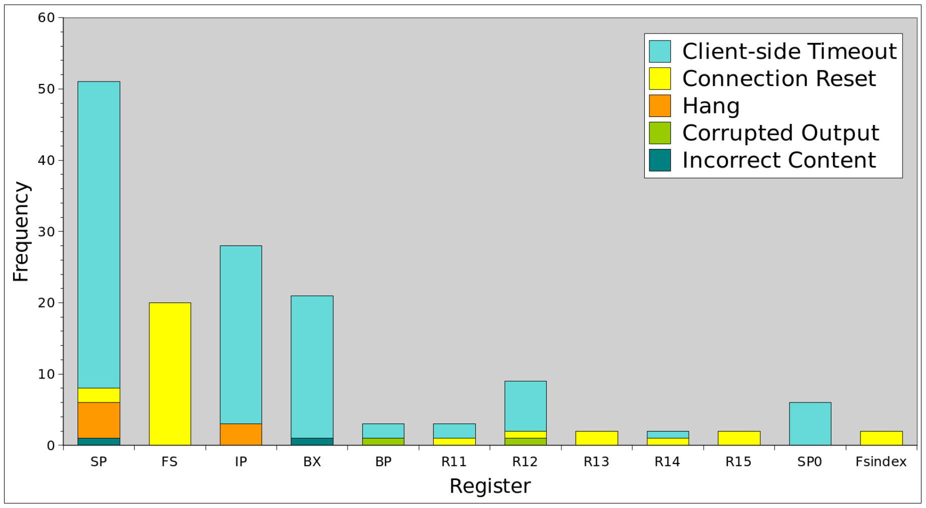

Figure 11: Distribution of failure modes across processor registers, for injections in application processes, in HVM mode

Interestingly the results were quite disparate between PV and HVM modes. Whereas in PV mode the most effective registers were IP, followed by BX. In HVM mode the SP register caused the majority of failures, followed by FS, IP and BX. To note the fact that in HVM the FS register caused solely *Connection Resets*. And that IP, BX and SP were more prone to causing *Client-side timeouts*. Finally, the range of registers that produced failures is bigger in HVM than in PV mode (12 vs 8 registers).

The IP (Instruction Pointer) register is a essential register in any computer system and has the role of controlling the execution flow. It is therefore no surprise that when this register is corrupted it can produce unexpected results. The SP (Stack Pointer) register is another key part of every computer system and holds the top of the stack, a essential data structure. If this value is corrupted the stack will point to a incorrect memory address. The BX (Base Register) register is a general-purpose register that usually contains a data pointer used for indirect addressing. Furthermore, Linux uses this register for storing parameters when performing a system call. The FS register is a segment register that has no fixed purpose, but is available for the Operating Systems to use as they see fit.

Figure 12 shows the distribution of the error manifestation latency, for both HVM and PV modes. The x-axis uses a logarithmic scale for representing time. From its analysis it is possible to understand that while some injections had an almost immediate effect, a considerable amount staid dormant during a few seconds. There is also an outlier that had a latency of 326 seconds.

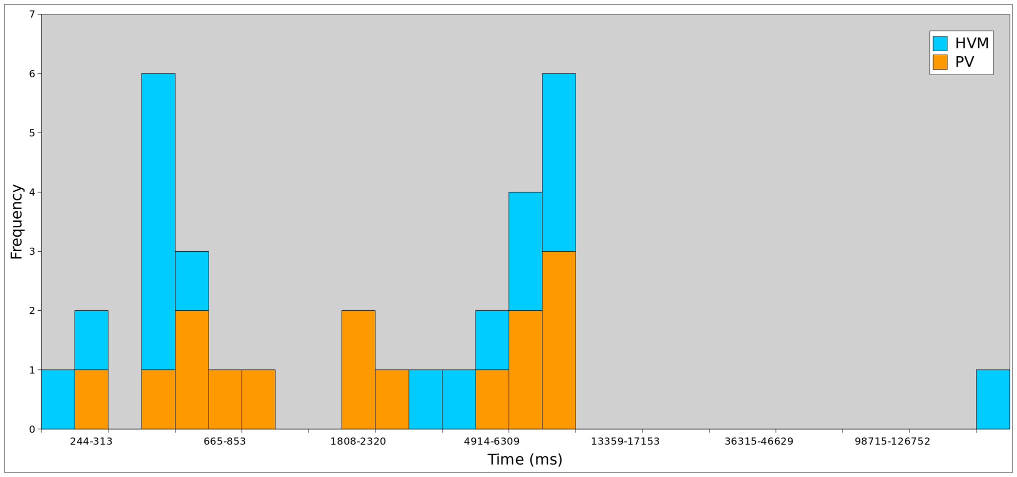

Figure 12: Manifestation latency, in PV and HVM

5.4.2 Injection in OS's processes inside VM

After injecting bitflips into the webserver's processes we moved on to processes spawned by the kernel (kernelspace processes). These kind of processes are more privileged than userspace and therefore might have a bigger impact in the behaviour of the virtual machine both to the outside and inside (interaction with other VMs). Given the relative high amount of kernel processes active in a system we opted by choosing one random kernel process to inject every time. Table 6 presents the results for this scenario when executing the VM under HVM mode, and Table 7 presents when under PV mode.

Virtual machine 1 (Fault VM)		Virtual machine 2 (Control VM)	
Incorrect content	0 (0%)	Incorrect content	0 (0%)
Corrupted output	0 (0%)	Corrupted output	0 (0%)
Connection reset	0 (0%)	Connection reset	0 (0%)
Client-side timeout	5 (1.0%)	Client-side timeout	0 (0%)
Hang	4 (0.8%)	Hang	0 (0%)
No effect	493 (98.2%)	No effect	502 (100%)
Hypervisor responsive	502 (100%)		

Table 6: Outcomes of fault injection targeting OS processes within a virtual machine, in HVM mode

Virtual machine 1 (Fault VM)		Virtual machine 2 (Control VM)	
Incorrect content	0 (0%)	Incorrect content	0 (0%)
Corrupted output	0 (0%)	Corrupted output	0 (0%)
Connection reset	0 (0%)	Connection reset	0 (0%)
Client-side timeout	2 (0.9%)	Client-side timeout	0 (0%)
Hang	1 (0.4%)	Hang	0 (0%)
No effect	225 (98.7%)	No effect	228 (100%)
Hypervisor responsive	228 (100%)		

Table 7: Outcomes of fault injection targeting OS processes within a virtual machine, in PV mode

When comparing both, and taking into consideration the results from previous campaigns, it can be noticed that there is no longer a discrepancy in terms of manifestation percentage between both virtualization modes. Furthermore, the percentage of manifestations to injections was very low, and their categorization was limited to Hangs and Client-side timeouts.

Once again an analysis of the distrbution of failure modes according to the register was performed for register bitflips in dom0, and the results are presented in Figure 13.

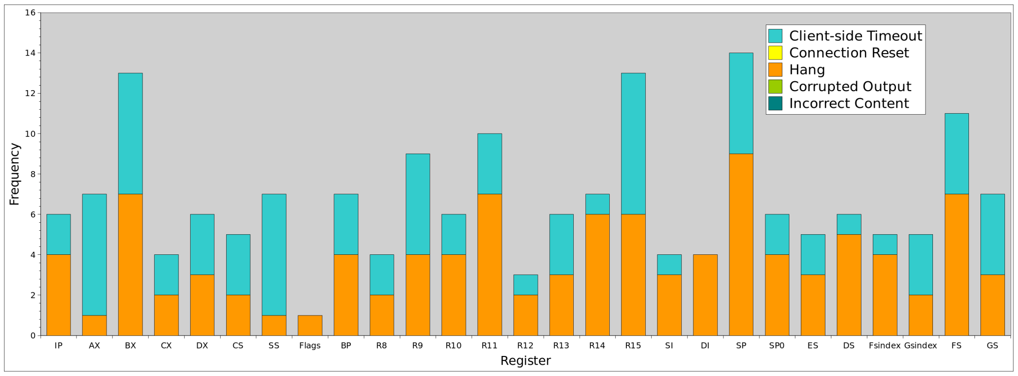

Figure 13: Distribution of failure modes across processor registers, for injections in the hypervisor dom0

Unlike what happened in the previous scenario, when performing register bitflips in dom0 every register produced an effect in the output, which caused the system to *Hang* or a *Client-side Timeout*.

5.4.3 Injection in Xen's processes in dom0

After injecting inside the guest virtual machines, we placed our attention into the privileged VM (dom0). In here we performed injections inside pre-chosen processes (both userspace and kernelspace) belonging to Xen. It should be noted that dom0 can only be virtualized using PV. The targeted processes were:

- oxenstored: storage scheme for storing VMs configurations;
- xenconsoled: daemon that provides control to the domains' consoles.
- qemu: used by Xen to enable the dom0 to access a virtual disk;
- xenwatchdogd: allows actions to be triggered when certain guest VMs are detected as crashed;
- xenbus (kernelspace): bus for interaction between domains;
- xenbus-frontend (kernelspace): frontend for xenbus;

The results for register bitflips are presented in Table 8.

Experiments	Hypervisor		Virtual machine 1		Virtual machine 2		Both VMs affected
965	Unresponsive	182	Incorrect content	0	Incorrect content	0	182 (100%)
			Corrupted output	0	Corrupted output	0	
			Connection reset	0	Connection reset	0	
			Client-side timeout	75	Client-side timeout	75	
			Hang	107	Hang	107	
			No effect	0	No effect	0	—
	Responsive	783	Incorrect content	0	Incorrect content	0	—
			Corrupted output	0	Corrupted output	0	
			Connection reset	0	Connection reset	0	
			Client-side timeout	0	Client-side timeout	0	
			Hang	0	Hang	0	
			No effect	783	No effect	783	—

Table 8: Outcomes of fault injection in processor registers, targeting the hypervisor in dom0

Of particular note is the fact that both guest VMs were equally affected by the injections. Furthermore all outcomes are classified as either *Hang* or *Client-side timeout*. However it should be noted that it is our belief, after a manual verification of the results, that the failures classified as *Client-side timeouts* are in reality *Hangs*. In fact due to the cool-off period of the experiment not being long enough, they were interrupted before being classified as *Hang*. These results prove the impact that kernelspace processes inside the privileged VM can have in the overall stability of the virtualization platform.

Due to the limited time that we had for performing the experiments we had to prioritize the most important and likely valuable scenarios first, which meant that memory injections would be paid much less attention than to register bitflips. However given the higher privilege that dom0 has compared to the guest virtual machines, we deemed that performing memory bitflips in this component would be of interest. Another reason for using this scenario is that it will give us data for later comparison, when we perform memory bitflips directly inside Xen memory. The results are displayed in Table 9.

Experiments	Hypervisor		Virtual machine 1		Virtual machine 2		Both VMs affected
102	Unresponsive	0	Incorrect content	0	Incorrect content	0	—
			Corrupted output	0	Corrupted output	0	
			Connection reset	0	Connection reset	0	
			Client-side timeout	0	Client-side timeout	0	
			Hang	0	Hang	0	
			No effect	0	No effect	0	—
	Responsive	102	Incorrect content	0	Incorrect content	0	—
			Corrupted output	0	Corrupted output	0	
			Connection reset	0	Connection reset	0	
			Client-side timeout	0	Client-side timeout	0	
			Hang	0	Hang	0	
			No effect	102	No effect	102	—

Table 9: Outcomes of fault injection in memory, targeting the hypervisor in dom0

The nonexistence of any failure makes it clear that a memory injection campaign will need to take much longer than a register injection campaign to obtain enough results.

5.4.4 Memory bitflips in Xen's memory

The results of performing memory bitflips inside Xen's memory (which is protected from access by any of the VMs) are presented in Table 10.

Experiments	Hypervisor		Virtual machine 1		Virtual machine 2		Both VMs affected
276	Unresponsive	1	Incorrect content	0	Incorrect content	0	1 (100%)
			Corrupted output	0	Corrupted output	0	
			Connection reset	0	Connection reset	0	
			Client-side timeout	0	Client-side timeout	0	
			Hang	1	Hang	1	
			No effect	0	No effect	0	—
	Responsive	275	Incorrect content	0	Incorrect content	0	—
			Corrupted output	0	Corrupted output	0	
			Connection reset	0	Connection reset	0	
			Client-side timeout	0	Client-side timeout	0	
			Hang	0	Hang	0	
			No effect	275	No effect	275	—

Table 10: Outcomes of fault injection in memory, targeting the hypervisor in Xen

The number of total injections performed during this campaign can be considered low when talking about memory bitflips, because of the inherently lower effect that a memory bitflip possess versus register bitflips. This fact reflects itself in only 1 bitflip having produced a perceptible failure (Hang).

5.4.5 Register bitflips in Xen's hypercalls

Finally, we performed register bitflips inside Xen's executable code. Given the extension of Xen's code we opted to target the code of the most often called hypercalls. Hypercalls have a very high number of calls and CPU time and cross the boundary between a VM and the hypervisor. Reasons that make it the best choice of location in where to inject the bitflips.

First, we used the *xentrace* tool, that allowed us to obtain a count of the number of calls that each hypercall had during a baseline execution of the workload. The results are presented in Table 11:

Hypercall ID	Hypercall Name	# Calls
1	mmu_update	2092
3	stack_switch	10232
13	multicall	552
14	update_va_mapping	90
17	xen_version	517
23	iret	31758
24	vcpu_op	17666
25	set_segment_base	2488
26	mmuext_op	2157
29	sched_op	6323
32	evtchn_op	1458
33	physdev_op	2724
35	sysctl	1

Table 11: Profiling of hypercall usage

The results show that the most often called hypercall was *iret*, and therefore we chose to target this hypercall in particular. From the information we managed to compile from online sources [33], this hypercall can be used to switch between user and kernel mode by a guest VM.

Given the very big range of possible permutations of line number, register, mask and injection time it was impossible to exhaustively test every combination. For this reason a smaller range of combinations that could be executed in a realistic timeframe was chosen.

These combinations were calculated by the following Python code, which returns 2176 different combinations, taking around 22 days for the fault injection campaign to complete:

```
# Line where payload will be injected (we jump 6 in 6 lines)
for n_linha in xrange(1, 92, 6):
  # Registers to be used
  for n_registo in xrange(0, len(registo_lista)):
    # Choose 2 random bits of to inject
    population = xrange(0, 63)
    registos_escolhidos = random.sample(population, 2)
    for n_bit in registos_escolhidos:
      # Miliseconds to wait since the computer has started until ↩
          injection is performed
      for n_tempo in xrange(220000, 300000, 20000):
```

It should be noted that due to an overlapse the range of tested bits ends at the 63rd bit, and not the 64th as was supposed. This is not a major problem because it is easy to perform the missing runs, however there was not enough time to do this task before the deadline of this thesis.

In Table 12 the results are presented.

Experiments	Hypervisor		Virtual machine 1		Virtual machine 2		Both VMs affected
2176	Unresponsive	94	Incorrect content	0	Incorrect content	0	94 (100%)
			Corrupted output	0	Corrupted output	0	
			Connection reset	0	Connection reset	0	
			Client-side timeout	18	Client-side timeout	18	
			Hang	76	Hang	76	
			No effect	0	No effect	0	—
	Responsive	2082	Incorrect content	60	Incorrect content	48	352 (16.9%)
			Corrupted output	0	Corrupted output	0	
			Connection reset	0	Connection reset	0	
			Client-side timeout	146	Client-side timeout	142	
			Hang	215	Hang	216	
			No effect	1661	No effect	1676	—

Table 12: Outcomes of fault injection in registers, targeting one hypercall in Xen

One point worth mentioning is the fact that there were no manifestations of *Corrupted Output* or *Connection Reset*. The hypervisor was sometimes capable of maintaining its responsive state, even if one or more of the VMs had been classified as *Hanged*.

In Figure 14 we can analyze which registers caused manifestations and in which VMs they had an effect. It is clear that there are a few registers that provoke more manifestations than the rest, namely RIP, RSP and RBX. We can also refer RAX, RDX, RBP, R12 and R14, that had a lower yet still significant impact. It appears that there were no major discrepancies between VMs, apart from a small deviation which is not unexpected.

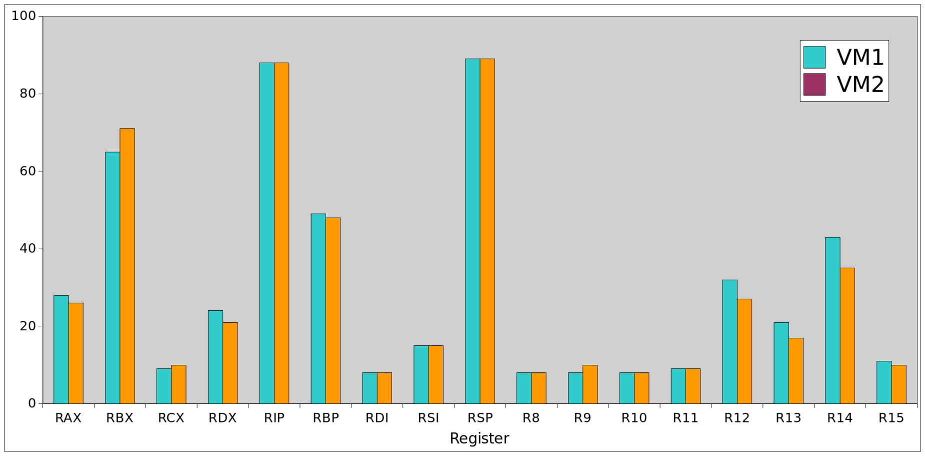

Figure 14: Distribution of manifestations across processor registers, for injections in Xen

Another interesting and useful way to look at these results is by analyzing which bit produced the biggest impact in all the experiments. However when parsing through this data we arrived at the conclusion that the distribution of the bits was not as uniform as we had hoped for. In fact, the distribution had an average of 34,53, a median of 32 and a standard deviation of 11,38. In Figure 15 we can see this distribution in a graphical manner.

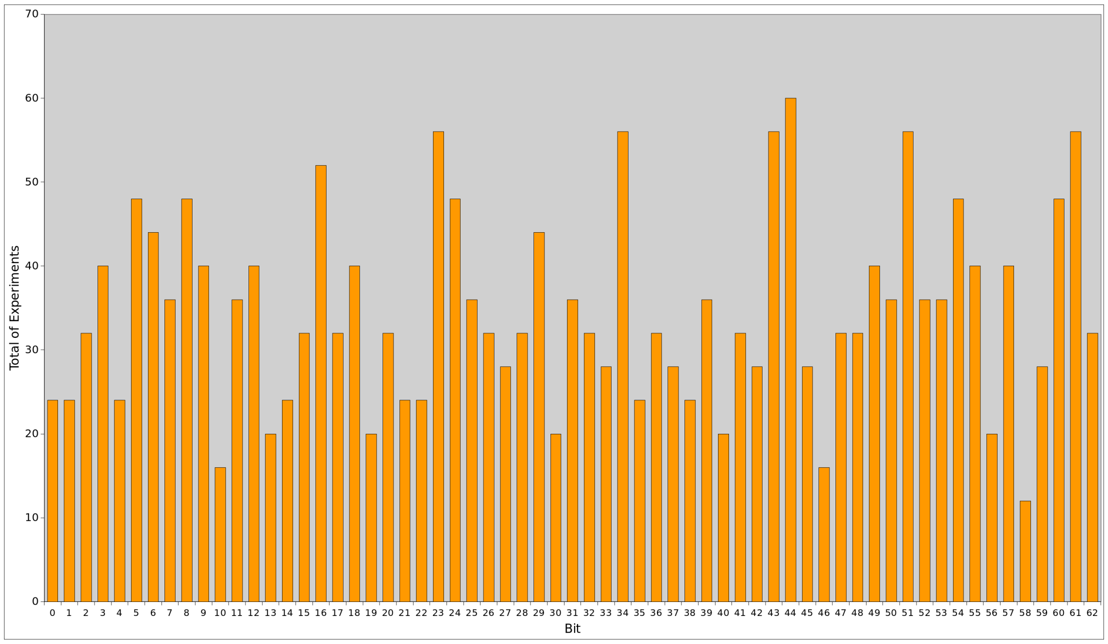

Figure 15: Distribution of the flipped bit, for injections in Xen code

In Figure 16 we assess the impact that each bit of the registers had when provoking a manifestation. Instead of using absolute values we are displaying percentage due to the previously stated reason. As can be seen, there were bits that had significantly more effect than others, bits that had completely no effect, namely bit 10 and 17, and others that only had a small effect in one of the VMs, namely 4, 29, 40 and 52.

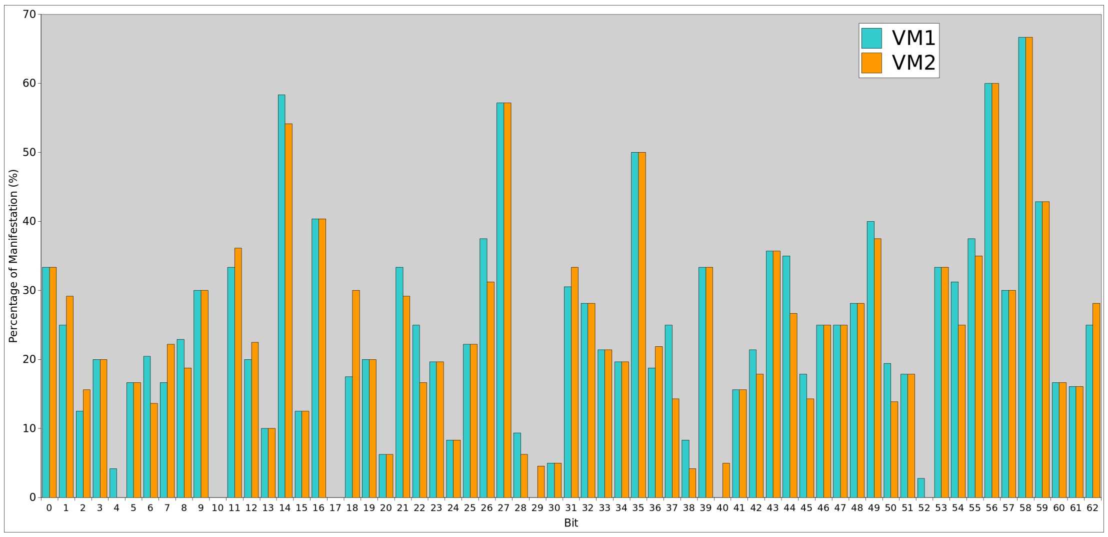

Figure 16: Distribution of manifestation percentage according to flipped bit, for injections in Xen

Another factor that was varied along this campaign was the time at which the injection would take place. In Figure 17 the amount of manifestations for each of the 4 timepoints is shown. The deviation among them is particularly low, which allows us to conclude that the time of injection has little to no impact in the outcome, at least for the 4 tested timepoints.

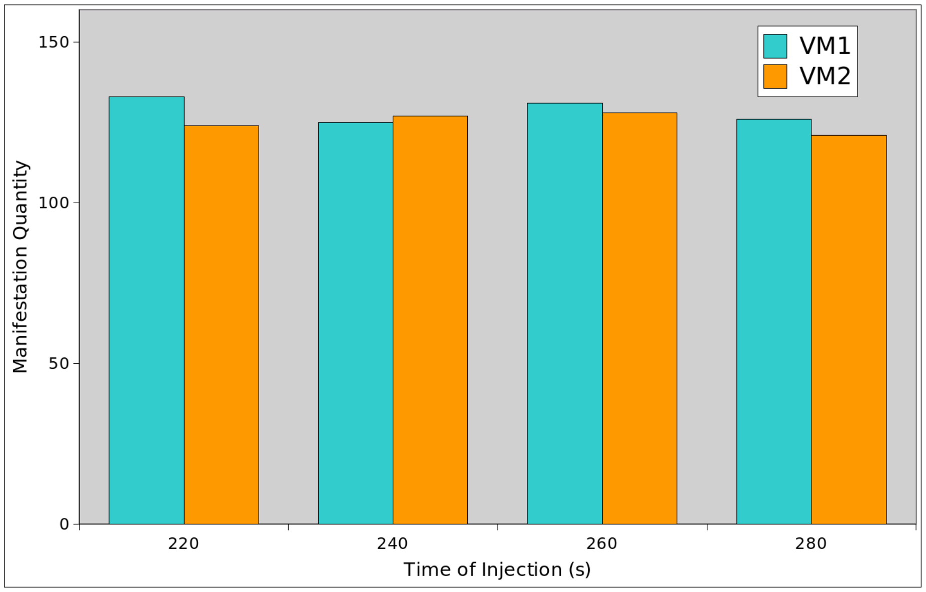

Figure 17: Distribution of manifestations according to time of injection, for injections in Xen

5.5 Conclusion

The results from the various campaigns have allowed us to better understand how a virtualized system reacts to the presence of a soft-error, in particular register and memory bitflips. By injecting in different areas of the system, namely inside the userspace and kernelspace applications of the guest VM, inside applications of the privileged guest and directly into the hypervisor, we were able to have a good coverage of the system and obtain valuable knowledge of how each area behaves. In summary:

- Memory injections are slower to produce a failure than register injections, and therefore their campaigns need an higher number of samples before useful results can be extracted.
- Injecting directly into the hypervisor can have a big impact, but not all the failure modes had expression, solely Hang, Wrong Content and Client-Side Timeout. On the other hand, injecting in the privileged VM (dom0) still has a significant impact, but only one failure mode is really present: Hang. Injecting in the guest VMs will produce all the 5 different failure modes but with a lower probability. Furthermore we also noted some discrepancies between the two virtualization modes: PV and HVM.
- Different registers have wildly different impact in the behaviour of the system, independently of the component where we inject. Across every campaign we can name IP, SP and BX as the 3 highest-impact registers.

- We never managed to break Xen's isolation layer, failure in one guest VM does not have impact in others. But a successful injection in dom0 will always affect both guest VMs and cause the hypervisor to become unresponsive. Meanwhile, an injection in Xen's hypercall can affect both guest VMs, but not forcefully.

All this information is a very useful contribution to the field, and can base the design of fault tolerance mechanisms for this kind of systems, by allowing more attention to be paid to more critical and sensitive areas of the system (e.g., more critical registers).

6 Conclusion and Future Work

In this section a overview of the work performed during this thesis is provided, followed by a review of the future work that will take place in the months following the end of this Master thesis.

6.1 Global Vision

During this thesis we performed extensive work in the area of evaluating and benchmarking virtualized systems for the cloud. Namely by conducting experiment campaigns using fault-injection tools by us developed, in which we assess the effect of a soft-fault inside a virtualized system using Xen hypervisor.

We also devised a watchdog system capable of detecting and restoring an unresponsive virtualized system, even in the event of the hypervisor being completely unresponsive, by using a technological feature existent in some Intel CPU's. In order to assess the effectiveness of this system, we benchmarked the vanilla system sans watchdog against the improved system using the watchdog, and concluded that our watchdog timer is effective in its purpose. However the biggest contribution of this thesis were the 3 fault injection tools that we developed and from now on can be used in future research projects. In fact, the first injection tool has already been used by a group of students to evaluate the durability of data in MySQL and Cassandra under the presence of register bitflips.

Using the work and knowledge obtained during this thesis, two articles were written and submitted to the *11th European Dependable Computing Conference - Dependability in Practice*, one of the articles was accepted with distinction and classified as a runner-up to the Best Article Prize.

By the end of this thesis we can safely feel satisfied with the work performed and confident of our contributions to this area.

6.2 Future Work

As we completed the work of this thesis, new opportunities for further development arose. Some of those opportunities could not fit inside the schedule of this thesis, and therefore will be completed in the following months.

- Finish the experiments of injecting register bitflips inside Xen's hypercalls, by increasing the breadth of hypercalls being targeted, and compare the results between directly injecting in Xen's code versus injecting in the privileged VM.
- Reproduce the work here presented in one of the main cloud service providers.
- The approach researched during this thesis can be used to guide the design of a fault tolerance mechanism capable of preventing crashes and covering a large portion of problems with a low cost (performance overhead).

Appendices

A Assembly code of *iret* hypercall

```
do_iret:
.LFB406:
    .loc 1 291 0
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
.LBB346:
.LBB347:
    .loc 2 29 0
    movq    $-32768, %rax
.LBE347:
.LBE346:
    .loc 1 296 0
    movl    $72, %edx
.LBB350:
.LBB348:
    .loc 2 29 0
#APP
# 29 "/run/media/root/b0b5c925-c631-4902-8a08-54↩
   ebadb18fe3/xen-4.4.1/xen/include/asm/current.h" 1
    and %rsp,%rax
# 0 "" 2
.LVL199:
#NO_APP
.LBE348:
.LBE350:
    .loc 1 291 0
    pushq   %rbx
    .cfi_def_cfa_offset 24
    .cfi_offset 3, -24
.LBB351:
.LBB349:
    .loc 2 30 0
    leaq    32536(%rax), %rbx
.LVL200:
.LBE349:
.LBE351:
    .loc 1 291 0
    subq    $88, %rsp
    .cfi_def_cfa_offset 112
    .loc 1 296 0
    movq    152(%rbx), %rsi
    .loc 1 294 0
    movq    32744(%rax), %rbp
.LVL201:
    .loc 1 296 0
    leaq    8(%rsp), %rdi
    call    copy_from_user
.LVL202:
    testq   %rax, %rax
    jne .L155
    .loc 1 305 0
    movq    48(%rsp), %rax
    movq    %rax, %rdx
    andl    $3, %edx
    cmpq    $3, %rdx
    je  .L156
.L146:
    .loc 1 317 0
    orl $3, %eax
    .loc 1 316 0
    movq    40(%rsp), %rdx
    .loc 1 317 0
    movw    %ax, 136(%rbx)
    .loc 1 318 0
    movq    56(%rsp), %rax
    .loc 1 316 0
    movq    %rdx, 128(%rbx)
    .loc 1 319 0
    movq    %rax, %rdx
    andq    $-143873, %rdx
    orb $2, %dh
```

```
	movq	%rdx, 144(%rbx)
	.loc 1 320 0
	movq	64(%rsp), %rdx
	movq	%rdx, 152(%rbx)
	.loc 1 321 0
	movzwl	72(%rsp), %edx
	orl	$3, %edx
	.loc 1 323 0
	testb	$1, 33(%rsp)
	.loc 1 321 0
	movw	%dx, 160(%rbx)
	.loc 1 323 0
	je	.L157
.L148:
	.loc 1 331 0
	movq	8(%rbp), %rcx
	shrq	$9, %rax
	.loc 1 333 0
	movq	%rbp, %rdi
	.loc 1 331 0
	xorq	$1, %rax
	andl	$1, %eax
	leaq	1(%rcx), %rdx
	movb	%al, (%rdx)
	.loc 1 333 0
	call	async_exception_cleanup
.LVL203:
	.loc 1 336 0
	movq	8(%rsp), %rax
.L151:
	.loc 1 342 0
	addq	$88, %rsp
	.cfi_remember_state
	.cfi_def_cfa_offset 24
	popq	%rbx
	.cfi_restore 3
	.cfi_def_cfa_offset 16
.LVL204:
	popq	%rbp
	.cfi_restore 6
	.cfi_def_cfa_offset 8
.LVL205:
	ret
.LVL206:
	.p2align 4,,10
	.p2align 3
```

```
.L157:
    .cfi_restore_state
    .loc 1 326 0
    movq    16(%rsp), %rax
    .loc 1 325 0
    andl    $-257, 124(%rbx)
    .loc 1 326 0
    movq    %rax, 48(%rbx)
    .loc 1 327 0
    movq    24(%rsp), %rax
    movq    %rax, 88(%rbx)
    movq    56(%rsp), %rax
    jmp .L148
    .p2align 4,,10
    .p2align 3
.L156:
    .loc 1 307 0
    cmpq    $0, 2560(%rbp)
    je  .L158
    .loc 1 313 0
    movq    %rbp, %rdi
    call    toggle_guest_mode
.LVL207:
    movq    48(%rsp), %rax
    jmp .L146
.L155:
    .loc 1 299 0
    call    current_domain_id
.LVL208:
    leaq    .LC27(%rip), %rsi
    leaq    .LC28(%rip), %rdi
    movl    %eax, %ecx
    movl    $300, %edx
    xorl    %eax, %eax
    call    printk
.LVL209:
.L145:
    .loc 1 339 0
    call    current_domain_id
.LVL210:
    leaq    .LC27(%rip), %rsi
    leaq    .LC30(%rip), %rdi
    movl    %eax, %ecx
    movl    $339, %edx
    xorl    %eax, %eax
    call    printk
```

```
.LVL211:
    .loc 1 340 0
    leaq    .LC27(%rip), %rsi
    leaq    .LC31(%rip), %rdi
    xorl    %eax, %eax
    movl    $340, %edx
    call    printk
.LVL212:
    movq    16(%rbp), %rdi
    call    __domain_crash
.LVL213:
    .loc 1 341 0
    xorl    %eax, %eax
    jmp .L151
.L158:
    .loc 1 309 0
    call    current_domain_id
.LVL214:
    leaq    .LC27(%rip), %rsi
    leaq    .LC29(%rip), %rdi
    movl    %eax, %ecx
    movl    $310, %edx
    xorl    %eax, %eax
    call    printk
.LVL215:
    .loc 1 311 0
    jmp .L145
    .cfi_endproc
```

References

[1] *ARK — Intel Core i7-4770 Processor (8M Cache, up to 3.90 GHz).* `http://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz`. Accessed: 2015-06-20.

[2] Algirdas Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: *Dependable and Secure Computing, IEEE Transactions on* 1.1 (2004), pp. 11–33.

[3] *AWS — Amazon — Instance Types.* `http://aws.amazon.com/ec2/instance-types/`. Accessed: 2015-06-20.

[4] Raul Barbosa et al. "Fault injection". In: *Resilience Assessment and Evaluation of Computing Systems.* Springer Berlin Heidelberg, 2012, pp. 263–281.

[5] James H. Barton et al. "Fault injection experiments using FIAT". In: *Computers, IEEE Transactions on* 39.4 (1990), pp. 575–582.

[6] Anton Beloglazov et al. "A taxonomy and survey of energy-efficient data centers and cloud computing systems". In: *Advances in Computers* 82.2 (2011), pp. 47–111.

[7] Carsten Binnig et al. "How is the weather tomorrow?: towards a benchmark for the cloud". In: *Proceedings of the Second International Workshop on Testing Database Systems.* ACM. 2009, p. 9.

[8] João Carreira, Henrique Madeira, and João Gabriel Silva. "Xception: A technique for the experimental evaluation of dependability in modern computers". In: *Software Engineering, IEEE Transactions on* 24.2 (1998), pp. 125–136.

[9] David Chisnall. *The Definitive Guide to the Xen Hypervisor.* Prentice Hall, 2007. Chap. 1.1. ISBN: 013234971X.

[10] Jörgen Christmansson and Ram Chillarege. "Generation of an error set that emulates software faults based on field data". In: *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on.* IEEE. 1996, pp. 304–313.

[11] *Citrix and Amazon Web Services.* `http://www.citrix.com/global-partners/amazon-web-services/overview.html`. Accessed: 2015-06-20.

[12] Mark Coggin. *Red Hat Enterprise Linux — The original cloud operating system.* Tech. rep. 10935437_0413. Red Hat, 2013.

[13] *Comparison of Platform Virtualization software - Wikipedia, the free encyclopedia.* `https://en.wikipedia.org/wiki/Comparison_of_platform_virtualization_software`. Accessed: 2015-06-20.

[14] *debugfs - Wikipedia, the free encyclopedia.* `http://en.wikipedia.org/wiki/Debugfs`. Accessed: 2015-01-17.

[15] Jim Gray. *Benchmark handbook: for database and transaction processing systems.* Morgan Kaufmann Publishers Inc., 1992.

[16] *How to read/write files within a Linux kernel module?* `http://stackoverflow.com/questions/1184274/how-to-read-write-files-within-a-linux-kernel-module/`. Accessed: 2015-06-20.

[17] Karl Huppler. "The art of building a good benchmark". In: *Performance Evaluation and Benchmarking.* Springer, 2009, pp. 18–30.

[18] *IFIP WG10.4 on Dependable Computing and Fault Tolerance.* `http://www.dependability.org/wg10.4/`. Accessed: 2015-01-17.

[19] *Injecting faults into the kernel.* `http://www.lwn.net/Articles/209257/`. Accessed: 2015-01-17.

[20] "Intel® Active Management Technology Overview". In: (2008).

[21] *Intel Designs Custom AWS CPU for Fastest EC2 Instances Ever.* `http://www.datacenterknowledge.com/archives/2014/11/13/intel-designs-custom-aws-cpu-for-fastest-ec2-instances-ever/`. Accessed: 2015-06-20.

[22] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. "FERRARI: A flexible software-based fault and error injection system". In: *Computers, IEEE Transactions on* 44.2 (1995), pp. 248–260.

[23] W-I Kao, Ravishankar K. Iyer, and Dong Tang. "FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults". In: *Software Engineering, IEEE Transactions on* 19.11 (1993), pp. 1105–1118.

[24] *kder - A Framework for Dynamic Analysis of Linux Kernel Modules.* `https://code.google.com/p/kedr/`. Accessed: 2015-01-17.

[25] *KVM.* `http://www.linux-kvm.org/page/Main_Page`. Accessed: 2015-06-20.

[26] Parag K Lala. *Fault tolerant and fault testable hardware design.* Prentice-Hall, 1985.

[27] Michael Le, Andrew Gallagher, and Yuval Tamir. "Challenges and Opportunities with Fault Injection in Virtualized Systems". In: *1st Int. Workshop on Virtualization Performance: Analysis, Characterization, and Tools.* 2008.

[28] *Less is More in the New Xen Project 4.5 Release — Xen Project Blog.* `https://blog.xenproject.org/2015/01/15/less-is-more-in-the-new-xen-project-4-5-release/`. Accessed: 2015-01-23.

[29] Robert Love. *Linux Kernel Development (3rd Edition).* Addison-Wesley Professional, 2010. Chap. 7. ISBN: 0672329468.

[30] Henrique Madeira, Diamantino Costa, and Marco Vieira. "On the emulation of software faults by software fault injection". In: *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on.* IEEE. 2000, pp. 417–426.

[31] Peter Mell and Tim Grance. “The NIST definition of cloud computing”. In: (2011).

[32] Michał Mosdorf and J Sosnowski. “Fault injection in embedded systems using GNU Debugger”. In: *Pomiary, Automatyka, Kontrola* 57 (2011), pp. 825–827.

[33] Jun Nakajima et al. “X86-64 XenLinux: architecture, implementation, and optimizations”. In: *Linux Symposium*. 2006, p. 173.

[34] *OpenStack User Survey Insights: November 2014*. `http://superuser.openstack.org/articles/openstack-user-survey-insights-november-2014`. Accessed: 2015-01-18.

[35] *Oracle VM VirtualBox*. `https://www.virtualbox.org/`. Accessed: 2015-06-20.

[36] *Project Roadmap*. `http://xenserver.org/overview-xenserver-open-source-virtualization/project-roadmap.html`. Accessed: 2015-01-18.

[37] Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. “Towards understanding the effects of intermittent hardware faults on programs”. In: *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on.* IEEE. 2010, pp. 101–106.

[38] Joanna Rutkowska and Alexander Tereshkin. “Bluepilling the xen hypervisor”. In: *Black Hat USA* (2008).

[39] Li Tan et al. “Investigating the Interplay between Energy Efficiency and Resilience in High Performance Computing”. In: ().

[40] Timothy K Tsai and Ravishankar K Iyer. *Measuring fault tolerance with the FTAPE fault injection tool*. Springer, 1995.

[41] *Virtualization*. `http://www.xenproject.org/users/virtualization.html`. Accessed: 2015-06-20.

[42] *vSphere ESXi Bare-Metal Hypervisor*. `http://www.vmware.com/products/esxi-and-esx/overview`. Accessed: 2015-06-20.

[43] *Windows Server 2012 R2 and Windows Server 2012*. `https://technet.microsoft.com/library/hh801901.aspx`. Accessed: 2015-06-20.

[44] Rafal Wojtczuk. “Subverting the Xen hypervisor”. In: *Black Hat USA 2008* (2008).