# ucXception: A Framework for Evaluating Dependability of Software Systems

Pedro David Almeida, Frederico Cerveira*, Raul Barbosa, and Henrique Madeira
University of Coimbra, CISUC, DEI
{pdsda@student., fmduarte@, rbarbosa@, henrique@}dei.uc.pt
*corresponding author

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

***Keywords*—*fault injection tool, fault injection, software faults, hardware faults, fault model, failure model***

## I. Introduction

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

Hardware faults, particularly transient ones (i.e., soft errors), are on the rise due to the increase in the number of hardware components [2], [3], allied to the miniaturization of microprocessors [4], [5], [6], [7] and the usage of energy saving techniques [7] (e.g., dynamic voltage and frequency scaling [8]). At the same time, evermore complex software systems are prone to residual software faults (i.e., bugs) that escape the software testing phase [9]. Software fault rates have been shown to be related with the total amount of source code lines, number of changed lines, and complexity of the system [10], [11], which are software features that have increased dramatically in recent years, with the consequent rise in the number of deployed bugs. Therefore, the evaluation of dependability and its attributes is of utmost importance, not only to critical systems but also to any software system.

To evaluate dependability, it is often necessary to perform *fault injection* (FI), which is a well-established technique that accelerates the process of fault activation in software systems by deliberately emulating faults in a system. Through the usage of fault injection, the generation of failure data is greatly accelerated, producing in a few weeks the amount of data that would otherwise have taken years. Despite fault injection having been widely used for several decades, the reality is that researchers and practitioners often develop their own fault injection tools from scratch, since there is a limited number of these tools in the public domain that are capable of being easily applied to different types of systems, supporting multiple fault models or possessing a low learning curve.

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

ucXception has been made open-source and can be found at https://github.com/ucx-code/ucXception. When compared with existing fault injection tools, ucXception is one of the few projects that natively supports fault injection in virtualized and cloud computing systems, features a graphical user interface and supports injection of both transient hardware faults and software faults out-of-the-box.

Thanks to its ability to integrate new components and tools, the experiments described in this paper employed ucXception to investigate whether failure models (i.e., actual failure/wrong outputs of components instead of injection of faults) are a valid alternative to fault models when performing fault injection. In other words, we researched whether injecting failures (e.g., node crash, process crash, process hang) yields results similar to those obtained using representative fault models (e.g., single bit-flip in CPU registers) in less time. The experiment focused on a popular cloud computing setup based around Openstack and used a workload composed by common operations that cloud administrators perform. A new fault injection tool was developed to inject failures, more precisely process crashes, and was integrated into ucXception. The results suggest that failure models can be used to accelerate campaigns, however the resulting failures appear to differ from those obtained when performing fault injection using fault models, recommending some care with the representativeness of experiments that use

2693-9177/22/$31.00 ©2022 IEEE
DOI 10.1109/QRS57517.2022.00063

failures modes.

The contributions of this paper include:

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

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

## II. Framework description

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

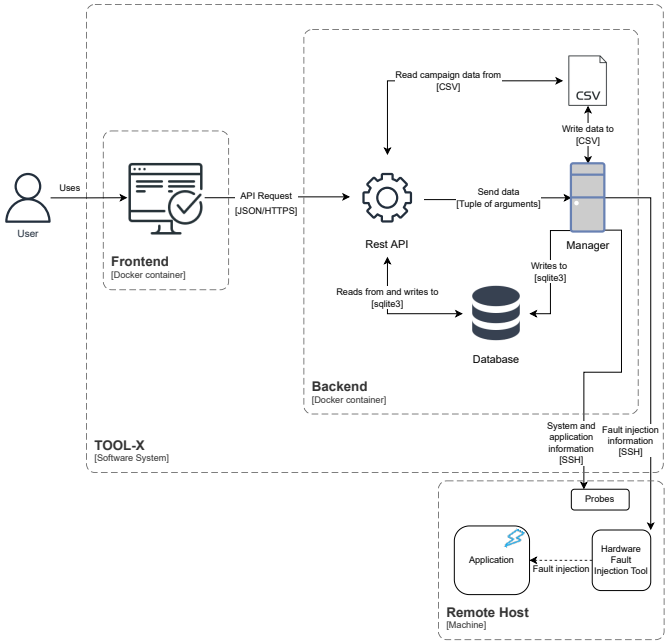


Figure 1. Architecture of ucXception.

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

The Backend module consists of two software components: the Manager and a REST API. The Manager is the heart of the framework and is the software component responsible for executing the fault injection campaign and storing its results. The Backend module spawns multiple instances of the Manager process, one for each campaign being executed. The REST API exposes the functionalities of the Manager to the Frontend module. The Backend module also encompasses one database, where the information about the users and their campaigns is kept, as well as multiple CSV files that contain the results of each campaign. The Backend module is self-sufficient and can be used without the Frontend module. The language adopted for the development of the Manager was Python 3.8, due to its simplicity and due to the range of libraries that are available. The technology used to develop the REST API was Flask and the database was SQLite, since the framework is aimed at smaller groups, for example research groups, hence a simpler and lighter engine was chosen.

### A. Frontend & Graphical User Interface

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

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

The first sub-module (Authentication) includes the functionalities that deal with account management and authentication. In this sub-module, 4 different pages are available: login, create account, recover password and change password. Although authentication is not a feature commonly found in other fault injection frameworks, it was added as it enables multiple users to share the framework seamlessly (e.g., multiple researchers of the same research center, students in a software quality/dependability course).

After performing the login, the user is presented with the menu page, which contains the list of the user's campaigns along with a brief description of their state. The user can search and filter campaigns by their name, execution name and campaign type, as well as varying the number of campaigns to display per page.

When a campaign has finished, the user can view a brief analysis of the results. The statistics page, shown in Figure 2, contains a statistical analysis of the campaign results, including

information regarding run duration, failure percentage, incorrect content percentage, among others. Furthermore, a line graph displays the evolution of the overall percentage of crash and incorrect data failures as the number of runs increases, thus indicating whether the results have converged or not.

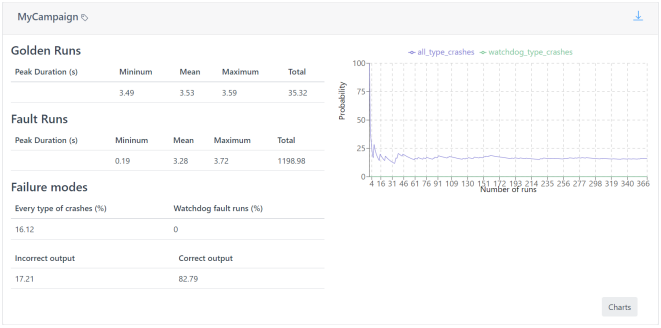


Figure 2. Statistics page.

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

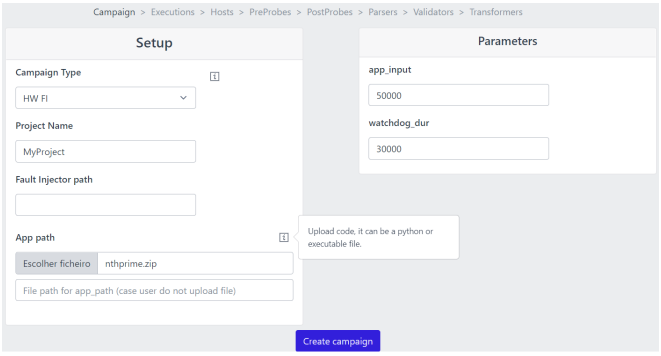


Figure 3. Create campaign page.

### B. Manager

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

- `Campaign` - A campaign consists in a set of runs according to a specific configuration (which is stored in a database). Different runs can have different parameters, e.g., some runs may perform injection while others will not (golden runs).
- `Run` - A run represents a single execution of the experiment flow defined in the campaign configuration, using the given run- and campaign-specific parameter values.
- `Watchdog` - A watchdog is used to monitor the execution time of a run and to ensure that it does not extend over the user-defined alloted time. If the run is taking too long and since it may even never end (e.g., the application has entered an infinite loop), the watchdog will kill the workload application and record the occurrence.
- `Probe` - A probe represents an application that will be launched for the duration of the run and has the purpose of monitoring and storing information relative to the system or application being evaluated. Probes can be subdivided into pre- and post- probes, according to whether they are launched before or after the workload has started. Pre-probes usually collect system-wide metrics, whereas post-probes monitor specific processes, hence the need for post-probes to be launched after the workload.
- `Fault injection tool` - The fault injection tool implements a specific fault model (e.g., software faults, single bit-flip for emulating transient HW faults) and allows the emulation of faults according to that model.
- `Validators` - The validators are small pieces of code, usually Python functions, that inspect the results obtained during a run and verify acceptance conditions. If a validator fails (e.g., fault injection was not successful) then the data for that run will not be written to disk.
- `Parsers` - A parser reads the results of the run (e.g., output of the fault injection tool, workload output) and converts them into a more useful and compact format. The various parsers write always to the results CSV file.
- `Transformers` - Transformers are similar to parsers, since both receive raw input and convert it into a processed output, but whereas parsers store their output in the results CSV file, the output from transformers is stored as individual files in the run's own result's folder. Transformers are mostly used to convert the raw output of the probes into a more manageable format, e.g., converting a binary file originating from a resource monitoring probe into a CSV file where each row represents a time step and each column represents a monitored resource.

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

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

1) *Launch pre-probes* - The pre-probes are launched and start to monitor their targets.
2) *Launch workload* - The workload is started. The user must programmatically define this step, possibly using the available utility functions.
3) *Launch post-probes* - The post-probes are launched. Normally the post-probes require the PID (or similar information) of the processes that they will monitor.
4) *Launch fault injection tool* - The fault injection tool (faultload) is launched. Per run only one injection is performed (unless explicitly modified in the code), in order to avoid a previous injection influencing future injections and thereby skewing the results. Although the fault injection tool is launched at this point, the fault may only be injected at a later moment, as the tool itself can have its own triggering mechanism. Unless modified, the values passed to the fault injection tool are randomly chosen from pre-defined valid ranges that will determine the actual type of fault injected in each run.
5) *Peak loop* - During this phase the workload executes, and the fault injection will take place at some point during its course. A watchdog process is launched with a configured pre-determined amount of time, if the workload does not finish within the alloted time, then the watchdog will forcefully kill the workload and the fault injection tool (if required). Otherwise this phase ends as soon as the workload process terminates.
6) *Post finish* - Usually consists, at least, in stopping the probes, but may include other user-defined operations that should be executed right after the workload has ended.
7) *Extract data* - Extracts the data from the probes and stores them in the run's results folder.
8) *Launch transformers* - Launches the transformers that will read the stored data and convert it to another format, which will once again be stored in the run's results folder.
9) *Launch parsers* - Launches the parsers that will produce the output is stored in the main results CSV.
10) *Launch validators* - The last step consists in validating the results as to ensure their correctness.

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

### C. Components

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

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

With regards to post-probes, only a single probe is available:

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

In terms of parsers, the following are available:

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

- *Current folder parser* - Minimalistic parser that just stores the path of the results folder of the current run.

Concerning transformers, the following are available:

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

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

### D. Fault Injectors

ucXception comes equipped with three fault injection tools that implement different fault models. There are two different fault injection tools for emulating hardware faults, which focus on different types of systems, and a tool for emulating software faults. Moreover, other fault injection tools can be integrated into the framework.

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

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

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

The moment of injection is always temporarily triggered, but there is support for two ways of setting this trigger: *timeout* and *deadline*. In *timeout* mode the user specifies how many milliseconds the tool should after it is launched and before it performs the bit-flip. Whereas in *deadline* mode, the user specifies a UNIX timestamp (including milliseconds) which defines the desired moment of injection and which the tool will attempt to obey as closely as possible. Localization-based triggering, i.e. triggering the fault whenever a certain instruction is executed, is not currently supported, as this tool's approach is not the best candidate to support such a triggering mechanism. The flow of the this fault injection tool is hence as shown in Figure 4.

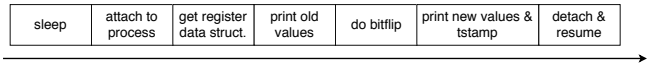


Figure 4. Flow of ucXception's HW fault injection tool for Linux-based systems.

*2) Hardware faults in virtualized systems:* A separate fault injection tool capable of emulating hardware faults was created specifically for use in virtualized systems. It is capable of injecting faults in any application running inside a VM, including a hypervisor as long as nested virtualization is used (i.e., the hypervisor being targeted is executed inside a VM).

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

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

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

Furthermore, this tool is capable of filtering the application that is targeted for injection by looking at the value in the *rip* register (which points to the next instruction to be executed)

and only performing injection whenever the *rip* is inside a user-defined range. This functionality can be specially useful if one wishes to perform fault injection that affects solely the hypervisor (or solely the non-hypervisor code) running in a VM, as there is a well established division between the virtual memory addresses assigned to the hypervisor, to the operating system and to the userspace applications.

Figure 5 presents the expected usage scenario for this tool.

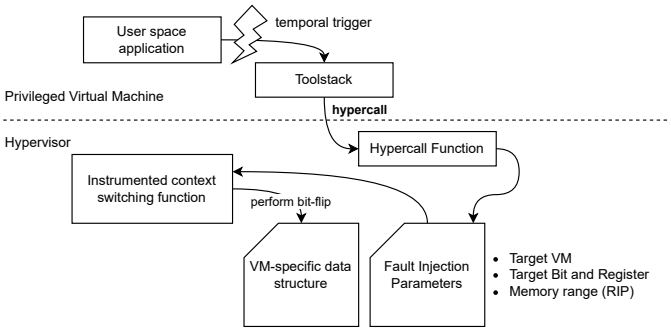


Figure 5. Flow of ucXception's tool for virtualized systems.

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

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

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

The injector is written in Java, using the Eclipse CDT plugin, which is the plugin that supports C/C++ programming in Eclipse. The fault injector takes as input the source code and CDT performs lexical and syntactic analysis to produce the corresponding abstract syntax tree. The fault injector then searches the tree to identify the nodes in which faults can be injected. For each possible location/fault type pair, the tree is modified accordingly and the resulting program is then converted back into source code representation. Then, a call to the `diff` tool generates a `patch file` for each fault that can be injected.

### *E. Containerization*

One of the unique advantages of ucXception is its easy and quick installation process, which is accomplished through the Docker containerisation technology. Docker is an open platform for building, running, and managing containers on servers and the cloud. Docker revolves around two basic concepts: images and containers. An image is a read-only template with instructions for creating a container, and a container is a runnable instance of an image. To publicly share images, Docker provides a repository, called Docker Hub. ucXception consists of two images (one for the Frontend module and another for the Backend module) which are publicly available in Docker Hub at https://hub.docker.com/r/ucxcode/ucxception/tags.

From Docker Hub, an administrator interested in installing the framework can pull and launch the containers using a few commands, however some configuration is required. The administrator must define an IP address, a port and two environment variables, which are:

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

If the fault injection campaigns will take place in remote systems (i.e., a system other than where the framework is installed), then a private/public keypair should be shared across the framework and every remote system. If using the images published in Docker Hub, then a pre-defined keypair already exists, but a new keypair can be used by manually creating the Docker image. Fortunately, creating new images in Docker is straightforward and requires only editing the commands of the Dockerfile to include the new keypair.

## III. Evaluating Failure Models for Campaign Acceleration

Fault injection using fault models has been widely used for evaluating the dependability of systems and to validate fault tolerance mechanisms. However, despite being effective, it is a slow process because many faults do not have any effect (i.e., do not cause any visible failure in the target system). Fault injection using failure models can, hypothetically, be able of reproducing the same results, with similar levels of accuracy and representativeness, but at a fraction of the time and cost.

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

For the experiments we will use the ucXception framework and take advantage of its extensibility to integrate a new fault injection tool that uses a failure model and a new parser. We chose an experimental setup representing a cloud deployment and opted to use Openstack, a cloud operating system, as the target system for the experiments.

### A. Setup

In order to run the experiments a physical setup was configured, which consisted of a single machine equipped with an Intel(R) Core(TM) i7-4770 CPU, 16 GB of RAM, and two 7200 RPM hard drives. The machine was running the Xen 4.1.1 hypervisor and the VMs were using Linux 4.14.89.

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

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

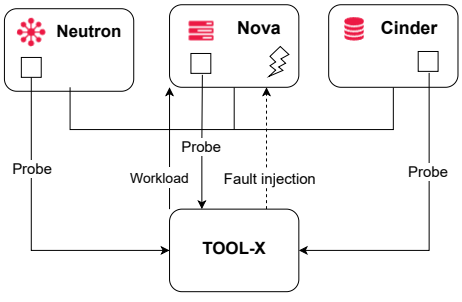


Figure 6. Experimental Setup.

Figure 6 illustrates how the setup is configured and how fault collection and injection is performed. ucXception executes the workload and launch the probes for each service installed. While the workload is being executed, the ucXception manages the fault injection process.

### B. Workload

The workload consists of several types of requests made to Openstack, which represent some of the most common operations that a system administrator might perform, such as listing, creating and deleting flavors or instances. The workload follows a sequential flow which performs, in total, 11 operations (listed from T1 to T11), as follows:

1) List all flavors (i.e., the resource configurations that can be used by the virtual machines) - T1, T4, T11
2) List all instances (i.e., virtual machines) - T2, T6, T10
3) Create a new flavor with a certain configuration - T3
4) Create a new instance - T5
5) Delete a flavor - T9
6) Delete an instance - T8
7) Check instance is responsive - T7

The workload is simplistic and some functionalities are tested by more than one operation. In total, the workload takes on average 130 seconds to finish.

### C. Injection process

Regarding the injection process, the *what, where and when* to inject was decided based on the goals of our study. Since the objective of the study was to compare injecting faults and failures, two models were used. Regarding the fault model, the single bit-flip fault model was used. This model emulates soft errors affecting the CPU register file directly or other CPU components (buses, ALU, FPU, etc.) indirectly. Both the bit and register are chosen randomly according to an uniform distribution, once per run, and a specific process (nova-api) of Openstack is targeted.

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

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

One of the various Openstack-related services of the Nova VM was chosen to be the target, as Nova is the central element of the setup. In the future we plan to conduct similar experiments in the other services of Nova and of the other Openstack components.

As the workload takes about 130 seconds to execute, the injection time was set to the range between ]10,100[ seconds, chosen randomly. The first 10 seconds correspond to the warmup and the last 30 seconds are the cooldown, which allow the injected faults to manifest and cause failures. To avoid hangs in the experiments, a watchdog was defined with a 200 seconds timeout (>1.5x the average duration).

### D. Failure detection and classification

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

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

### E. Results

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

Figure 7 shows a comparison between the amount of operations that failed in a single run for fault and failure models. It should be noted that the Y-axis is zoomed in between 0 and 6%. An analysis of the results concludes that not only failure models cause more failures, the generated failures also affected more operations, on average.

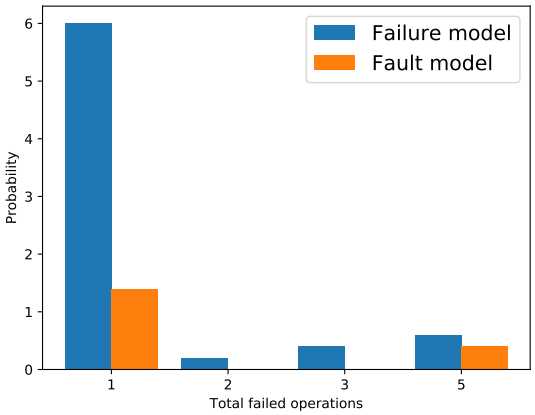


Figure 7. Failed operations per run for both models.

Figure 8 shows the probability of an error occurring in each of the various operations of the workload. Once again the Y-axis is zoomed in. The graph shows that operation T7 has a higher failure probability, when using both fault and failure models, which can be explained by being a more sensitive operation or by the moment of injection. The two first and last operations did not experience any failure likely due to the warmup and cooldown periods.

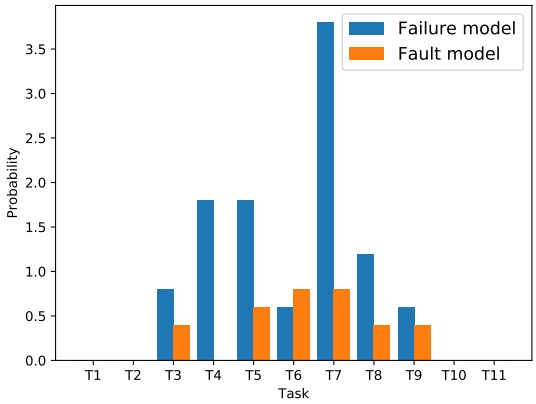


Figure 8. Failures per operation for both models.

To measure how the failure probability evolves when using failure models, Figure 9 plots the distance between the average failure probability obtained in each run when using failure models against the final failure probability obtained using fault models (i.e., 3.4%). It can be seen that the distance begins to stabilize after around the 200th run, but tends to slightly increase as the runs increase.

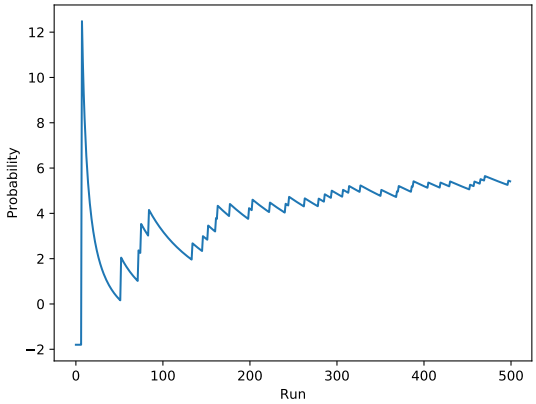


Figure 9. Distance between failure model and oracle.

### F. Observations & Limitations

Despite the exploratory nature of this experimental evaluation, some observations can be made. However more experiments are needed to reach acceptable confidence. One observation is that the usage of failure models produces more failures than the usage of fault models, thus accelerating this type of experimental campaigns. This is to be expected after all, but carries implications to practice. Namely, if the objective of a campaign is to quantify the failure modes and their probabilities of a system, the usage of failure models should be complemented with a correction factor, otherwise the obtained

probabilities will be higher than what they really are. On the other hand, if the objective of a campaign is solely to collect failure data or to validate a fault tolerance mechanism, then failure models appear to be a valid and more efficient choice.

Another observation is that the failures obtained when using failure models appear to differ from the failures generated by injecting faults. This can be seen in Figure 8, where the distribution of failures across different operations appears to not follow the same pattern. For example, failure models caused failures that strongly affected operation T7 whereas fault models did not. Another example is in operation T4, which experienced failures when using failure models but never when using fault models.

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

Even though limited, the results support carrying out further experiments in this topic and show how the extensibility of ucXception can be an advantage when researching topics that fall out from the more traditional use cases associated with fault injection.

## IV. Related Work

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

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

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

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

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

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

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

Goofi-2 [26] is capable of using both hardware-implemented and software-implemented techniques to emulate transient hardware faults in CPU registers and memory using single and multiple bit-flips. It supports three different fault injection techniques: Nexus-based, exception-based and instrumentation-based. All these techniques are in one way or another dependent on features of the underlying hardware.

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

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

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

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

## V. Conclusion

In this paper, we presented ucXception, an open-source framework for orchestrating and conducting fault injection

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

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

## ACKNOWLEDGMENT

This work is funded by the FCT - Foundation for Science and Technology, I.P./MCTES through national funds (PIDDAC), within the scope of CISUC R&D Unit - UIDB/00326/2020 or project code UIDP/00326/2020. This work is also supported by the FCT within project ECSEL/0018/2019 and the ECSEL Joint Undertaking (JU) under grant agreement No 876852. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, Turkey. Disclaimer: The views expressed in this document are the sole responsibility of the authors and do not necessarily reflect the views or position of the European Commission. The authors, the VALU3S Consortium, and the ECSEL JU are not responsible for the use which might be made of the information contained in here.

## REFERENCES

[1] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan 2004.

[2] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner, "United states data center energy usage report," 2016.

[3] J. Koomey, "Growth in data center electricity use 2005 to 2010," *A report by Analytical Press, completed at the request of The New York Times*, vol. 9, 2011.

[4] P. Hazucha and C. Svensson, "Impact of cmos technology scaling on the atmospheric neutron soft error rate," *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2586–2594, Dec 2000.

[5] S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, vol. 19, no. 4, pp. 23–29, July 1999.

[6] R. Baumann, "The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction," in *Digest. International Electron Devices Meeting,*, Dec 2002, pp. 329–332.

[7] V. Chandra and R. Aitken, "Impact of technology and voltage scaling on the soft error susceptibility in nanoscale cmos," in *2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, Oct 2008, pp. 114–122.

[8] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott, "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling," in *Proceedings Eighth International Symposium on High Performance Computer Architecture*, Feb 2002, pp. 29–40.

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

[10] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88.

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

[12] I. Corporation, "Intel PCM," https://github.com/opcm/pcm, 2019, accessed: 2019-02-01.

[13] S. Godard, "SYSSTAT," http://sebastien.godard.pagesperso-orange.fr/, 2019, accessed: 2019-02-01.

[14] Tcpdump, "TCPDump," https://www.tcpdump.org/, 2019, accessed: 2019-02-01.

[15] D. Faggioli, "Tracing with xentrace and xenalyze," https://blog. xenproject. org/2012/09/27/tracing-with-xentrace-and-xenalyze, 2012.

[16] R. Johansson, *On Single Event Phenomena in Microprocessors*, 1993.

[17] G. L. Ries, G. S. Choi, and R. K. Iyer, "Device-level transient fault modeling," in *Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing*, June 1994, pp. 86–94.

[18] R. Barbosa, J. Karlsson, H. Madeira, and M. Vieira, *Fault Injection*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 263–281.

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

[20] J. Arlat, Y. Crouzet, and J. . Laprie, "Fault injection for dependability validation of fault-tolerant computing systems," in *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, June 1989, pp. 348–355.

[21] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Ysskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "Fiat - fault injection based automated testing environment," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.*, June 1995, pp. 394–.

[22] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: a flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 248–260, Feb 1995.

[23] W. . Kao, R. K. Iyer, and D. Tang, "Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1105–1118, Nov 1993.

[24] D. Costa, H. Madeira, J. Carreira, and J. G. Silva, *Xception™: A Software Implemented Fault Injection Tool*. Boston, MA: Springer US, 2003, pp. 125–139.

[25] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer, "Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*, 2000, pp. 91–100.

[26] D. Skarin, R. Barbosa, and J. Karlsson, "Goofi-2: A tool for experimental dependability assessment," in *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, June 2010, pp. 557–562.

[27] M. Le, A. Gallagher, and Y. Tamir, "Challenges and opportunities with fault injection in virtualized systems," in *1st Int. Workshop on Virtualization Performance: Analysis, Characterization, and Tools*. Citeseer, 2008.

[28] A. Thomas and K. Pattabiraman, "Llfi: An intermediate code level fault injector for soft computing applications," in *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.

[29] M. Cinque and A. Pecchia, "On the injection of hardware faults in virtualized multicore systems," *Journal of Parallel and Distributed Computing*, vol. 106, pp. 50 – 61, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731517300849

[30] C. Pham, D. Chen, Z. Kalbarczyk, and R. K. Iyer, "Cloudval: A framework for validation of virtualization environment in cloud infrastructure," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, 2011, pp. 189–196.