# Fault injection through binary rewriting during native execution on real hardware

*Diploma Thesis*

By

Konstantelias Ioannis

Department of Electrical and Computer Engineering
University of Thessaly

A dissertation submitted to the University of Thessaly in accordance with the requirements of the Diploma degree in the Department of Electrical and Computer Engineering.

Supervisors
Christos D. Antonopoulos, Assistant Professor
Nikolaos Bellas, Associate Professor

January 2016

# Περίληψη

Τα τελευταία χρόνια υπάρχει μία τάση προς την εύρεση τρόπων μείωσης της ενέργειας που καταναλώνει ένα υπολογιστικό σύστημα. Ένας τρόπος για να γίνει κάτι τέτοιο είναι να ρίξουμε την τάση τροφοδοσίας των κυκλωμάτων. Όμως, μία τέτοια ενέργεια μπορεί να αποβεί μοιραία για τη σταθερότητα του συστήματος. Είναι πολύ πιθανό να βρεθεί σε μία ασταθή κατάσταση. Από την άλλη πλευρά, η συνεχής βελτίωση των ψηφιακών συστημάτων μέσω της συρρίκνωσης του μεγέθους των τρανζίστορ, αυξάνει τις πιθανότητες για κατασκευή συστημάτων που περιέχουν ατέλειες ή τείνουν να παρουσιάζουν εσφαλμένη συμπεριφορά κάτω από κάποιες συνθήκες.

Υπάρχουν έρευνες για την ταυτοποίηση εφαρμογών οι οποίες είναι δεκτικές σε λάθη που μπορεί να εισαχθούν σε αυτές κατά τη διάρκεια εκτέλεσής τους. Όπως αναφέρθηκε, τα λάθη αυτά έχουν ρίζα προέλευσης το κομμάτι του υλικού ενός συστήματος. Συνήθως φαίνονται ως λάθη που συμβαίνουν στην αρχιτεκτονική κατάσταση του συστήματος. Έτσι έχουν αναπτυχθεί εφαρμογές οι οποίες τοποθετούν λάθη σε ένα λογισμικό, προσομοιώνοντας έτσι τέτοιες αστοχίες υλικού. Ένα λογισμικό το οποίο κάνει αυτή τη δουλειά είναι το GemFI των Γ. Τζιατζιούλη και Κ. Παρασύρη, το οποίο εισάγει λάθη σε διάφορα στάδια του pipeline μιας κεντρικής επεξεργαστικής μονάδας σε παράλληλες εφαρμογές. Το εργαλείο αυτό στηρίζεται στον προσομοιωτή Gem5. Στα θετικά αυτής της απόφασης συγκαταλέγεται το γεγονός ότι η εισαγωγή λαθών είναι αληθοφανής καθώς μπορεί να γίνει με μεγάλη λεπτομέρεια. Στα αρνητικά αυτής της επιλογής είναι η καθυστέρηση που εισάγει η εκτέλεση ενός προσομοιωτή μεγάλης ακρίβειας, ο οποίος τρέχει πάνω από το λειτουργικό σύστημα.

Η προσφορά της παρούσας διπλωματικής είναι η δημιουργία ενός λογισμικού εισαγωγής λαθών, το οποίο όμως δουλεύει με εφαρμογές οι οποίες εκτελούνται σε ένα πραγματικό σύστημα. Για να γίνει κάτι τέτοιο εφικτό, χρησιμοποιήθηκε η τεχνική της δυναμικής επανεγγραφής των εντολών της γλώσσας μηχανής του επεξεργαστή την ώρα που η εφαρμογή εκτελείται. Έτσι, κατά την εκτέλεση, εντοπίζεται σε ποιό σημείο θα μπει το λάθος, κι έτσι αλλάζει η εντολή που εκτελείται. Το θετικό με αυτή τη μέθοδο είναι ότι μπορεί να οδηγήσει σε γρηγορότερες εκτελέσεις πειραμάτων κι έτσι να επιταχυνθεί η εξαγωγή αποτελεσμάτων για εφαρμογές υπό εξέταση.

# Abstract

During the last years, there is a tendency to explore ways of decreasing the power supply needed for a computing system to work, which will result in lower power consumption. A way of doing this is through dropping the power supply voltage of the digital system. On the other hand, the need for system performance by shrinking the size of the transistor, leads to a state that the chips are getting denser and denser. Both of the described facts lead to only one result: The digital system may/will be found in an unstable state.

There is research going on in order to find applications that can produce acceptable results even if they execute on faulty hardware. In order to find out, there is a need for tools that simulate such faults and insert them in the application running, in order to evaluate its dependability. A fault injecting tool that injects faults in the architectural state of a processor is GemFI; a work of G. Tziatzioulis and K. Parasyris for their thesis. It is based on simulation based fault injection method. Although it provides fine-grained injection mechanisms, it also introduces a big run-time overhead because of the simulation based technique.

My contribution is to provide a fault injection tool that tries to eliminate the problem of the big run-time overhead. For this reason Dynamic Binary Instrumentation technique has been used in order to inject faults in applications at run-time, on real hardware.

# Dedication and acknowledgements

In the first place I want to thank my lead supervisor, Dr. Christos Antonopoulos, Dr. Nikolaos Bellas and PhD candidate Konstantinos Parasyris for their assistance and guidance for the completion of this work.

I would also like to thank my friends and my family that were by my side all these years.

To my family and my friends.

## Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: ..... DATE: .....

# Table of Contents

# List of Tables

# List of Figures

**Figure** **Page**

Chapter 1

# Introduction

During the past decades, an explosive improvement on the processor microchip technology has been observed. Processors are becoming denser and denser, because of the continuous, steady shrinking of the transistors. As a result, they are becoming faster, but the demands of power are also increasing in order to keep up with the constant need of performance improvements. Although everything seems to work well, a set of constraints block somehow various aspects of this train of improvements.

In the early years, single core processors existed. The only way to make them faster was to use more transistors. In order to achieve that on a chip with restrictions on the dimensions, the transistors must become smaller. Taking into account *Moore's law*[*], transistor count indeed was increasing on a chip. The first problem that emerged was the demands of power to supply the chips. With the rate of transistor increase, the demands would soon reach the ones of a nuclear reactor [1]. So the industry moved to parallel computing and thus, parallel processors in order to resolve the power problem and of course applications' execution time. So, *power consumption* is an important aspect to consider.

Nevertheless, chips are getting denser. Not only processor chips, but also memory chips etc. Denser chips result in hardware that is more possible to transition to an unstable, faulty state. So, the developers should take into account this possibility when building a system. In addition to hardware malfunction by accident, malevolent actions can be taken to drive it in a wanted unstable state, eg. exploitation of Dynamic Random Access Memory (DRAM) rowhammer bug to gain Linux superuser privileges [2]. For this reason, *reliability* and *dependability* of a system is another important thing to consider.

Taking into account the aspects discussed previously, there is a field of computer science that has to do with fault injection techniques. Fault injection in order to test the reliability and dependability of a system. In addition to that there is also the SCoRPiO project which is developing [sic] "a new computing

[*] Moore's law is the observation that the number of transistors in a dense integrated circuit doubles approximately every two years.

paradigm that exploits uncertainty to design systems that are energy-efficient and scale gracefully under hardware errors by operating below the nominal operating point, in a controlled way, without inducing massive or fatal errors".

So, this thesis builds a fault injection tool in order to test the reliability and dependability of various applications in case of a hardware malfunction. This work could then be used to find applications that work well under unstable states of hardware and thus, run them on a system that operates below the nominal point with significantly reduced power consumption.

The chapters included in this thesis are organized in three parts, as can be seen in table 1.1.

| | | |
|---|---|---|
| Pt. 1 | Chapter 1 | Introduction in Greek |
| | Chapter 2 | Introduction in English |
| | Chapter 3 | Fault models |
| Pt. 2 | Chapter 4 | Dynamic Binary Instrumentation |
| | Chapter 5 | Presentation of Intel Pin |
| | Chapter 6 | Our implementation |
| Pt. 3 | Chapter 7 | Experiments and results |
| | Chapter 8 | Conclusions |

Table 1.1: Chapter overview

Chapters 1 and 2 are introductory chapters that state the purpose of this thesis. Chapter 3 presents what fault injection is. Chapter 4 is an introduction to the dynamic binary instrumentation technique used for building the tool. Subsequently, chapter 5 describes Intel's dynamic binary instrumentation tool, Pin. In chapter 6 there is a detailed presentation of our implemented fault injector. Chapter 7 is about the experiments taken with our injector on specific applications. Finally, chapter 8 concludes the current thesis.

# Chapter 2

# Fault Injection

Fault injection was originally discovered to serve as a software or hardware testing tool to determine how a system is dependent from the parts that are injected. That is because a system may not produce the intended results. In order to understand the fundamentals of fault injection, one must know the difference between *fault*, *error* and *failure*:

| | |
|---|---|
| Fault | flaw that exists in a part of a system |
| Error | deviation from an acceptable result |
| Failure | abnormal behavior of the system |

Table 2.1: Fault injection terms

In addition to the table 2.1, an *error* is the result of a *fault* manifestation. Also if a system should take a specific action that it did not take, that is a *failure*.

In this chapter we will describe what are the fault categories and what are the types of fault injection that exist taking into account Ziade's *et al.* fault injection survey [3] and which ways we use to tamper some operations of the system we operate on.

## 2.1 Fault Categories

### 2.1.1 Hardware Faults

Hardware faults can be categorized by the duration of their effects when they happen, in *permanent* and *temporary*. Temporary faults can be categorized in *transient* and *intermittent*. In more detail:

- *Permanent faults* - exist due to a physical component damage. Component damage may occur because of aging, improper manufacture or misuse. This kind of faults can be corrected by replacing

the tampered component.

- *Transient faults* - occur due to environmental conditions such as power-line fluctuations, electromagnetic interference, or radiation (e.g. cosmic rays). They are more possible to be triggered than permanent faults. Also they are the hardest to detect.
- *Intermittent faults* - are triggered in intervals and are caused by unstable hardware or varying hardware states.

Replication of almost all the hardware faults is easy. There are dedicated hardware tools that can flip bits *ad hoc* at the pins of a chip, vary the power supply or bomb the system with heavy ions-methods, in order to achieve results of a hardware transient fault. There are also ways to emulate transient faults with software based techniques. A technique, that we used, is via binary instrumentation. For both permanent and transient faults there are methods to inject them from a VHDL design [4].

### 2.1.2 Software Faults

Software faults are the ones that can be found in the stages of the development life of a software system. A prone to errors software development process is the *waterfall model*. *Waterfall model* is the one that follows a sequential, downward style of development through the phases that can be seen in the colored rectangles in figure 2.1.2. If an error occurs in any of the phases, the development life is continued until the *verification* phase. If the testing in that phase fails, then the project is discarded. The errors could be caused due to faults that exist in any of the previous phases.

That said, software faults can be found in *requirements gathering and analysis*, *system design*, *implementation* and for sure *verification* and *system deployment* phases of the development life of a software. These faults can be categorized according to [3] in:

- *Function faults*: Incorrect or missing implementation that requires a design change to be corrected.
- *Algorithm faults*: Incorrect or missing implementation that can be fixed without the need of a design change.
- *Timing/serialization faults*: Missing or incorrect serialization of shared resources.
- *Checking fault*: Missing or incorrect validation of data, or incorrect loop, or incorrect conditional statement.
- *Assignment fault*: Values assigned incorrectly or not assigned.

## 2.2 Types of Fault Injection

This section describes in a concise manner what are the types of fault injection, alongside with their advantages and disadvantages. There is also a list of fault injection tools that exist for each category.

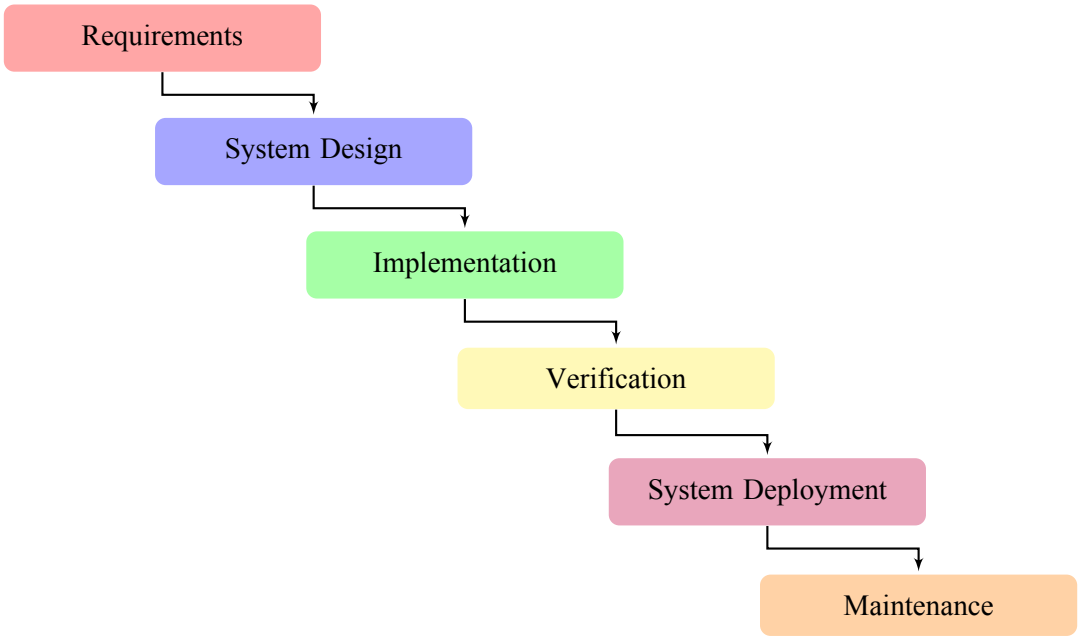



Figure 2.1: Software development process presented as a Waterfall model.

### 2.2.1 Hardware-based Fault Injection

Hardware-based fault injection methods can be classified in those that need or no contact. Examples of contact based methods are injection of the pins of an Integrated Circuit (IC) and power fluctuations. Without contact based methods are heavy ion radiation, electromagnetic interference and creation of high temperature conditions.

**Advantages**

- Hardware fault injection technique can access locations that are hard to be accessed by other means.
- This technique works well for the system which needs high time-resolution for hardware triggering and monitoring.
- Experiments are fast. They are near real execution time.

**Disadvantages**

- Hardware fault injection can introduce high risk of damage for the injected system.
- Limited set of injection points and limited set of injectable faults.
- The setup time for each experiment might, in fact, offset the time gained by the ability to perform the experiments in near real-time.
- Requires special-purpose hardware in order to perform the fault injection experiments.

- Limited observability and controllability.

**Tools**

- *RIFLE* - A pin-level fault injection system for dependability validation.
- *FOCUS* - A design automation environment used for analyzing a microprocessor-based jet-engine controller used in the Boeing 747 and 757 aircrafts.
- *FIST* - Employs both contact and contactless methods to create transient faults inside the target system.
- *MARS* - MARS system is a time-triggered, fault-tolerant, distributed system.

### 2.2.2 Software-based Fault Injection

Software-based fault injection methods can be used instead of hardware-based and achieve the results of the latter. With this technique, the register file, the memory and other components that could be tampered from hardware errors can be injected with faults.

**Advantages**

- This technique can be targeted to applications and operating systems, which is difficult to be done using hardware fault injection.
- Experiments can be run in near real-time, allowing for the possibility of running a large number of fault injection experiments.
- No need for special purpose hardware; low complexity, low development, low implementation cost.

**Disadvantages**

- Limited set of injection instants: At assembly instruction level, only.
- It cannot inject faults into locations that are inaccessible to software.
- It is very difficult to model *permanent faults*.

[3] states that a disadvantage is the fact that the source code must be modified in order to support fault injection mechanisms. That does not apply in our case because we used Dynamic Binary Instrumentation (DBI) techniques to achieve the needed results. So I do not list it as a disadvantage.

**Tools**

- *FIAT* - An automated real-time distributed accelerated fault injection environment.
- *XCEPTION* - Uses the advanced bugging and performance monitoring features present in many of today's modern processors to inject faults.

- *DOCTOR* - Integrated software fault injection environment allows injections into the CPU, memory and also network-communication faults.
- *NFTAPE* - The objective of NFTAPE is to support several different types of fault injection, providing the capability of targeting several heterogeneous systems concurrently.
- *PINFI* - A tool that uses DBI in order to inject faults at run-time.

### 2.2.3 Simulation-based Fault Injection

Simulation-based fault injection methods make use of a simulator that models the system under test. Usually the simulator is implemented in a Hardware Description Language (HDL), such as VHDL, Verilog and its family or in any other programming language. The work of K. Parasyris and G. Tziantzoulis [5] was the implementation of a simulation-based fault injection mechanism using and extending the cycle accurate simulator GEM5 [6].

**Advantages**

- Simulated fault injection can support all system abstraction levels.
- Not intrusive.
- Full control of both fault models and injection mechanisms.
- Does not require any special-purpose hardware.
- Maximum amount of observability and controllability.
- Able to model both transient and permanent faults.

**Disadvantages**

- Large development efforts.
- Time consuming.
- Accuracy of the results depends on the goodness of the model used.
- Model may not include any of the design faults that may be present in the real hardware.

**Tools**

- *VERIFY* - A tool that uses an extension of VHDL for describing faults correlated to a component, enabling hardware manufacturers, which provide the design libraries, to express their knowledge of the fault behavior of their components.
- *MEFISTO-C* - It uses the vantage optimum VHDL simulator and injects faults via simulator commands in variables and signals defined in the VHDL model.

- *HEARTLESS* - A hierarchical register-transfer-level fault-simulator for permanent and transient faults that was developed to simulate the fault behavior of complex sequential designs like processor cores.

## 2.3 Our choice

The fault injection tool that is developed in this thesis is categorized as software-based fault injection mechanism. As has been stated, opposed to simulation-based techniques results in faster fault injection campaigns and takes place on real hardware. In order to avoid the disadvantage that the source code of the application is needed we have chosen to use DBI. More details about DBI and its advantages will be given in the next section.

There is also another work on fault injection that is done using the framework of our choice, PINFI [7]. Although it is used for fault injection in the level of assembly, it was developed for a specific reason; to evaluate the results of another tool, LLFI [7, 8]. As a result, the tool is not general and could not fit in our preferences. However, some specific parts of the source code are used in our final implementation.

Chapter 3

# Dynamic Binary Instrumentation

We have seen in the previous chapter a list of methods used for fault injection. The method of our choice was driven by our need to inject faults during native execution in near real-time of execution. We also wanted to stay clean and not write a tool that requires from the target application program to know about it. In other words, no APIs that pollute the original source code, which in many cases is not available. So, we concluded on using software-based fault injection and particularly Dynamic Binary Instrumentation (DBI). In this chapter there will be an analysis of what DBI is.

## 3.1 Types of Analysis

### 3.1.1 Static vs Dynamic Program Analysis

Static analysis uses the information of the original code without executing it. It can achieve full code coverage but is not able to determine run-time specific data and information. This particular type of analysis is generally used in compilers (symbolic analysis, control flow graphs transformations, type checking etc.).

In contrast, dynamic analysis takes place at the run-time. Generally, it analyzes the code that results from the dynamic execution path of a given input. As a result, different inputs trigger different paths of the same code. On the good side, the fact that dynamic program analysis is more detailed makes it able to have access to all the fine information at run-time. Tools based on dynamic analysis are mainly computer security (taint analysis, reverse engineering, etc), profiling and debugging tools (cache performance, call chains, instruction mix etc).

### 3.1.2 Source vs Binary Analysis

Source analysis techniques include the source code of the program. They perform source analysis and extract information that have immediate relation to the source (e.g. compilers with control flow graphs). Source analysis often extract information mapped to granularities that a programming language introduce such as functions, statements, expressions and variables.

On the contrary, binary analysis is based on the object or executable code of the program. Binary analysis often extract information on granularities that are machine dependent such as images, sections, functions, traces, superblocks, basic blocks and instructions.

For what it's worth, source and binary based analysis techniques are usually used in the field of computer security. A chain of source analysis (along with penetration testing, fuzzing etc.) take place in order to find and repair all the easy-to-see vulnerabilities. After all this procedure and if need be (crucial parts of a program), binary analysis tools are ready to shoot. If the source code is not available, only binary analysis tools try to serve the job (reverse engineering).

## 3.2 Instrumentation

Instrumentation is the process of inspecting, analyzing and modifying the behavior of a computer program. In other words, given a program, the appliance of instrumentation on it involves fine-grain examination of the code to extract some low-level information (e.g. return value of a malloc). During this examination, an analysis can be performed using the low-level information in order to provide high-level information (e.g. memory leaks which is the list of mallocs not freed). Additionally, the instrumentation process includes addition/removal of code to/from the original code and/or modification of existing code.

### 3.2.1 Static vs Dynamic Instrumentation

Given the points above, the differences between these types of instrumentation are obvious:

- *Static instrumentation* is done on the code when the program is not in execution state. Specifically, it rewrites the code which then, can be executed.
- *Dynamic instrumentation* is performed at run-time of a program and usually involves an external program or tool.

### 3.2.2 Source vs Binary Instrumentation

As for the flavors of instrumentation:

- *Source instrumentation* takes place on the source code
- *Binary instrumentation* takes place on the object or executable code.

## 3.3 Dynamic Binary Instrumentation

In brief, DBI is the process of inspecting, analyzing and modifying the behavior of a running program regardless of the programming language used to create it.

### 3.3.1 Pros & Cons

In more detail, DBI doesn't require the source code of the application program. That means that it's agnostic of the programming language used to create it. Hence, there is no need for re-compilation or re-linking and also proprietary programs can be instrumented. Additionally, DBI can discover code at run-time and can be attached to already running processes. Due to the fact that only a binary file is needed, the whole DBI process can be monitored and debugged with the use of a debugger. Also, dynamic analysis grants access to run-time context info.

On the other hand, code coverage is a more difficult task to achieve with DBI. The most important disadvantage though, is that DBI adds spacial and timing overhead to the execution of the application program, which in many cases is impossible to measure it. The tricky part is the difficulty to determine for what portion of the overhead each part of the instrumentation is responsible.

### 3.3.2 Uses of DBI

DBI technique is greatly used in computer security. Also it is the main ingredient of many profilers. Also it is used by debugging tools and software testing tools.

## 3.4 Why DBI

We have chosen to use DBI as our method of injecting faults to a programs instructions because:

- we wanted to apply injections according to frequencies given by the user
- we wanted to test on programs from which we don't have the source code
- we did not want to hack the original source code of the programs for testing
- we wanted to inject assembly instructions in order to emulate architectural pipeline failures

## 3.5 DBI frameworks

A DBI framework is an Application Programming Interface (API) which gives the ability to its users to instrument a computer program. In most occasions the user of the API will have to write another program that does something on a target application. This program is usually called Dynamic Binary Analysis tool. Generally speaking, it attaches to the program to be instrumented and operates on it.

There are plenty of DBI frameworks and at first glance, either they can be categorized in those who work on an Intermediate Representation (IR) and those who work on the final binary code (architecture target assembly) or those that are suitable for heavy-weight instrumentation and those for light-weight. In the next chapter I will refer some of the popular DBI frameworks and briefly describe them before going deeply in the one that we used.

Chapter 4

# Intel® PIN

DBI frameworks are relatively popular among other instrumentation-technique frameworks for the reasons that we discussed in the previous chapter. In the current chapter we will take a glance at the most noteworthy DBI frameworks used and then describe the one of our choice, which is Pin, a tool developed by Intel.

## 4.1 DBI frameworks

A DBI framework can be categorized according to different metrics. One metric, proposed by the Pin team [9] is if the DBI framework is probe-based or jit-based. Probe-based tools insert trampolines in the code and when an instruction is met, the execution is transfered through these trampolines to the injected code. Pin team preferred to use the jit-based method in which the binary is dynamically compiled and instrumentation code (or calls to it) can be placed anywhere in the binary. Jit-based approach is better as it is more accurate. *Fine-grained* instrumentation is at question if using probe-based instrumentation because:

1. instrumentation is not transparent because original instructions in memory are overwritten by trampolines
2. on architectures where instruction sizes vary (i.e. x86), we cannot replace an instruction by a trampoline that occupies more bytes than the instruction itself because it will overwrite the following instruction
3. trampolines are implemented by one or more levels of branches, which can incur a significant performance overhead

In table 4.1 one can see a list of both jit and probe based DBI frameworks available:

| | |
|---|---|
| probe-based | Dyninst [10]<br>DTrace, [11] |
| jit-based | Vulcan, [12]<br>Diota [13]<br>DynamoRIO [14]<br>Pin [9]<br>Strata [15]<br>Valgrind [16] |

Table 4.1: Some DBI frameworks categorized by the way they insert instrumentation code.

### 4.1.1 Metrics for choice

Some other metrics for choosing one DBI framework over the other can be which granularities of instrumentation they provide or if they operate on Intermediate Representation (IR) or straight on binary. There are also heavy-weight vs light-weight DBI frameworks, where with first ones one can write tools that perform complex jobs relatively easy to the latter, but they lack in speed. A popular heavy-weight BDI framework is Valgrind [16]. Pin [9] is light-weight.

## 4.2 Pin

Pin uses just-in-time (JIT) compilation method to insert and optimize code. In contrast to the fashion of JIT technique that operates on bytecode, Pin operates on regular native code. The advantages of using Pin is that offers *programmable instrumentation*, it is *multiplatform*, *robust*, and targets *efficiency*.

### 4.2.1 Programmable instrumentation

Pin enables the user to write their own instrumentation programs, called **Pintools**. Pintools can be written in C/C++/assembly. The Application Programming Interface (API) provided is relatively simple to learn and hides the details of the Instruction Set Architecture (ISA) that the tool operates on. If need be, one can use an ISA specific API.

### 4.2.2 Multiplatform

Pin is built to run on Windows, Linux, OSX and Android. The architectures that supports are IA-32 and Intel64. Also there is support for Intel Xeon Phi but in Tech Preview version. Although in the first years had support for ARM, it is no longer supported.

### 4.2.3 Robust

Pin can instrument real-life multithreaded applications like databases (eg. MySQL), web-browsers (eg. Mozilla Firefox) and others. It also supports signals, exceptions, self modifying code etc.

### 4.2.4 Efficiency

As it has been mentioned, Pin relies on dynamic compilation. That means that the code does not need re-compilation. Optimizations like *trace linking*, *register re-allocation*, *thread-local register spilling* are performed. Also various compiler optimizations are applied on instrumentation code such as *inlining*, *variable liveness*.

## 4.3 Instrumenting with Pin

Pin instruments the code in a JIT fashion: When it encounters the first instruction of the executable, it generates new code for the *trace* that this instruction is head of.

> **NOTE:** A *trace* is a single entry, multiple exit sequence of instructions. It usually begins at the target of a taken branch and ends with an unconditional branch, including calls and returns. Figure 4.5(a) shows a trace example.

It then transfers control to this trace, which is almost identical to the original code. Pin guarantees that it will regain control when a branch exits the trace. After regaining control, Pin generates new code for the branch target and continues execution. Pin makes this procedure efficient by keeping the translated (instrumented) code in a cache-memory for further execution of the same sequence of instructions. So, firstly comes the instrumentation of the trace and then the execution of it. Figure 4.5(b) shows how Pin makes a copy of the trace in the code cache. In figure 4.5(c) Pin instruments the trace in the code cache and redirects the exit points to Pin. Finally in figure 4.5(d) Pin transfers control into code cache.

To put it differently, the whole process consists of two phases: *Instrumentation* and *Analysis*. Instrumentation and Analysis logic is programmed by the user in routines. *Instrumentation* routines define where instrumentation is inserted. E.g. before instruction, after a taken branch etc. They are executed when an instruction is being jitted. On the other hand, *Analysis* routines define what to do when instrumentation is activated. E.g. refresh opcode histogram, print instruction pointer etc. They occur every time an instruction is executed.

### 4.3.1 Pintool example

In figure A.2 the reader can see a simple example of a pintool that instruments every instruction of the application and traces the instruction addresses. The `main` function initializes the Pin, registers the function `Instruction` and starts the application. Function `Instruction` is called every time a new instruction is encountered or if that instruction is not in Pin's code cache. `INS_InsertCall` registers function `print_ip` to be called before (`IPOINT_BEFORE`) every instruction. *Instruction pointer*[*] value

[*] Instruction pointer is an alias to program counter. This term is extensively used in Pin so will be here.

is passed to it (`IARG_INST_PTR`). Finally `print_ip` prints the instruction pointer in the file, keeping a trace of the *dynamic instructions*[*].

Talking about arguments of analysis routines, here is a brief list:

- `IARG_INST_PTR` - Instruction pointer value
- `IARG_UINT32 <value>` - An integer value
- `IARG_REG_VALUE <register_name>` - Value of the register specified
- `IARG_BRANCH_TARGET_ADDR` - Target address of the branch specified
- `IARG_MEMORYOP_EA <memory_operator>` - Effective address of a memory operator
- ... and many more found in the Pin manual

### 4.3.2 Instrumentation Points

Instrumentation is relative to an instruction. As can be seen on table 4.2, instrumentation can be inserted *before* the execution of an instruction or *after*. Instrumenting before an instruction is a trivial process. On the other, when instrumenting after an instruction the pintool writer must take into account if that instruction has a *fall-through path* or not i.e. a *taken branch*, figure 4.1.

| Instrumentation points | |
|---|---|
| Before | `IPOINT_BEFORE` |
| After | `IPOINT_AFTER`<br>`IPOINT_TAKEN_BRANCH` |

Table 4.2: Instrumentation points

```
                cmp edx, esi
;IPOINT_BEFORE⟶
                jle <L1> ;IPOINT_TAKEN_BRANCH⟶ <L1>: ...
;IPOINT_AFTER ⟶
                mov edi, 0x1
```

Figure 4.1. Graphical representation of instrumentation points relatively to jle instruction.

[*] Dynamic instructions are called the application instructions that are executed given a particular input. On the other hand, static instructions are all the instructions that are included in a binary executable. It is analogous to the dynamic vs. static CFG. Run-time vs. Compile-time code.

### 4.3.3 Instrumentation Granularities

Instrumentation can be done at three different granularities:

- *Instruction*
- *Basic Block* - Single entry, single exit sequence of instructions.
- *Trace* - Single entry, multiple exits sequence of instructions. Stops with an unconditional branch.

In order to show how instrumentation applies to the different granularities we will see two different versions of a pintool that counts dynamic instruction. The demonstration will take place on the example code shown in figure 4.2.

The first implementation, that performs instrumentation on instruction granularity, figure 4.3 using function `Instruction`, registers the analysis function `docount0` before every dynamic instruction. The second implementation, using function `Trace` registers the analysis function `docount1` before every trace and passes to it the number of instructions that contains. The resulting instrumented code on the initial code from figure 4.2 can be seen in figures 4.4(a) and 4.4(b) respectively.

```
; Trace
  ; Basic Block 1
    sub edx, 0xff
    cmp edx, esi
  jle <L1>

   ;Basic Block 2
    mov edi, 0x1
    add eax, 0x10
  jmp <L2>
```

Figure 4.2. Graphical representation of instrumentation points relatively to jle instruction.

## 4.4 Code rewriting

In addition to instrumentation of the code, Pin API provides some functions and mechanism to also rewrite it. In this section, I will explain the basic tools that where used by the pintool I implemented to inject the code. Further details will reside in the next chapter.

### 4.4.1 The CONTEXT structure

During the process of a dynamic instrumentation it is necessary to know in every state and moment the architectural context of the executing process. Pin gives the ability to the pintool writer to extract

```
UINT64 icount = 0;

//=== Instruction granularity ============================================

// Analysis routine
void docount0() { icount++; }

// Instrumentation function
void Instruction(INS ins, void *v)
{
   INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount0, IARG_END);
}

//=== Trace granularity ==================================================

// Analysis routine
void docount1(INT32 c) { icount += c; }

// Instrumentation function
void Trace(TRACE trace, void *v)
{
   for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl)) {
      BBL_InsertCall(bbl, IPOINT_BEFORE,
         (AFUNPTR)docount1,
            IARG_UINT32, BBL_NumIns(bbl),
            IARG_END);
   }
}
```

Figure 4.3. A pintool that counts dynamic instructions. Full code can be found in Appendix A.

register values information through the `CONTEXT*` structure. It is a *handle* to the full register context of the application at a particular point in the execution.

`CONTEXT*` cannot be dereferenced by the pintool writer. The information can only be extracted via specific Pin API functions, e.g. `PIN_GetContextRegval()`. `CONTEXT*` is also passed by default to some Pin Callback functions, e.g. `PIN_AddThreadStartFunction`, `PIN_DefineTraceBuffer`, `PIN_AddContextChangeFunction`.

As it have been mentioned, Pin API provides a set of functions to operate on a `CONTEXT*` structure. Some of these involve getting and setting values of *architectural* and *scratch registers*. *Scratch registers* are the registers that Pin defines for use by the tools. Generally, they are virtual registers that exist to avoid using the real registers. Specifically, they are implemented in such way that some actions, like *memory operand rewriting* become a *fast atomic* process.

The pintool writer can pass `CONTEXT*` (`IARG_CONTEXT`) to an *analysis function*. This is a powerful feature, but it is time consuming. There is a method to make this process ˜4X times faster [17] and that

(a)

```
icount++
   sub edx, 0xff
icount++
   cmp edx, esi
icount++
   jle <L1>
icount++
   mov edi, 0x1
icount++
   add eax, 0x10
```

(b)

```
icount += 3 ; basic block 1
   sub edx, 0xff
   cmp edx, esi
   jle <L1>
icount += 2 ; basic block 2
   mov edi, 0x1
   add eax, 0x10

jmp <L2>
```

Figure 4.4. Dynamic instruction counting from two granularities: (a) Instruction, (b) Basic Block using Trace instrumentation.

is by passing a *constant* version of the `CONTEXT*` (`IARG_CONST_CONTEXT`) to the analysis functions, granted that the analysis functions will not set the context. Changes of the context in an analysis function will be available until the return of it. If the pintool writer wants to keep these changes upon the return of the analysis routine, they should call the Pin API function `PIN_ExecuteAt`, which sets the new context and restarts the execution from this point. This function never returns to the analysis function. Context manipulation was thoroughly used as a technique to insert faults in the original code. Every kind of error and the technique used will be explained with detail in the next chapter.

### 4.4.2 Other methods

`CONTEXT*` manipulation can be used to change register values at an execution point. Apart from this, the pintool writer can also delete instructions, change memory values and change control flow.

*Instruction deletion* can be done during the instrumentation phase with the function `INS_Delete`. It is a useful function if the pintool writer wants to emulate existing instructions. It is done by registering the analysis function that performs the emulation before the target instruction and then `INS_Delete` is

called. The actual instruction will not ever be executed, unless the pintool writer removes instrumentation.

Changing memory values can be done by directly writing to a valid memory address. For this reason, Pin provides `PIN_SafeCopy`. It does what a `memcpy` would do but also guarantees safe return to the caller even if the source or destination regions are inaccessible. If the pintool writer wants to change a memory reference, Pin provides a way to rewrite memory operands. `PIN_RewriteMemoryOperand`. This function replaces memory (read and/or write) operands from instructions with Pin's scratch registers. After this, the pintool writer can not only access that memory but also can change the memory these operands refer to.

Finally, the pintool writer can change the control flow of the program by using the functions that replace branch targets and then deleting the actual branch. These functions are `INS_InsertDirectJump` and `INS_InsertIndirectJump`.

In my implementation of fault injection tool, I do not really use these functions for injecting faults. And that is because they happen at instrumentation phase. The only one I make use of is the one that rewrites memory operands in association with scratch registers. The next chapter will refer to some of the Pin's functions alongside with the description of our fault-injection philosophy.

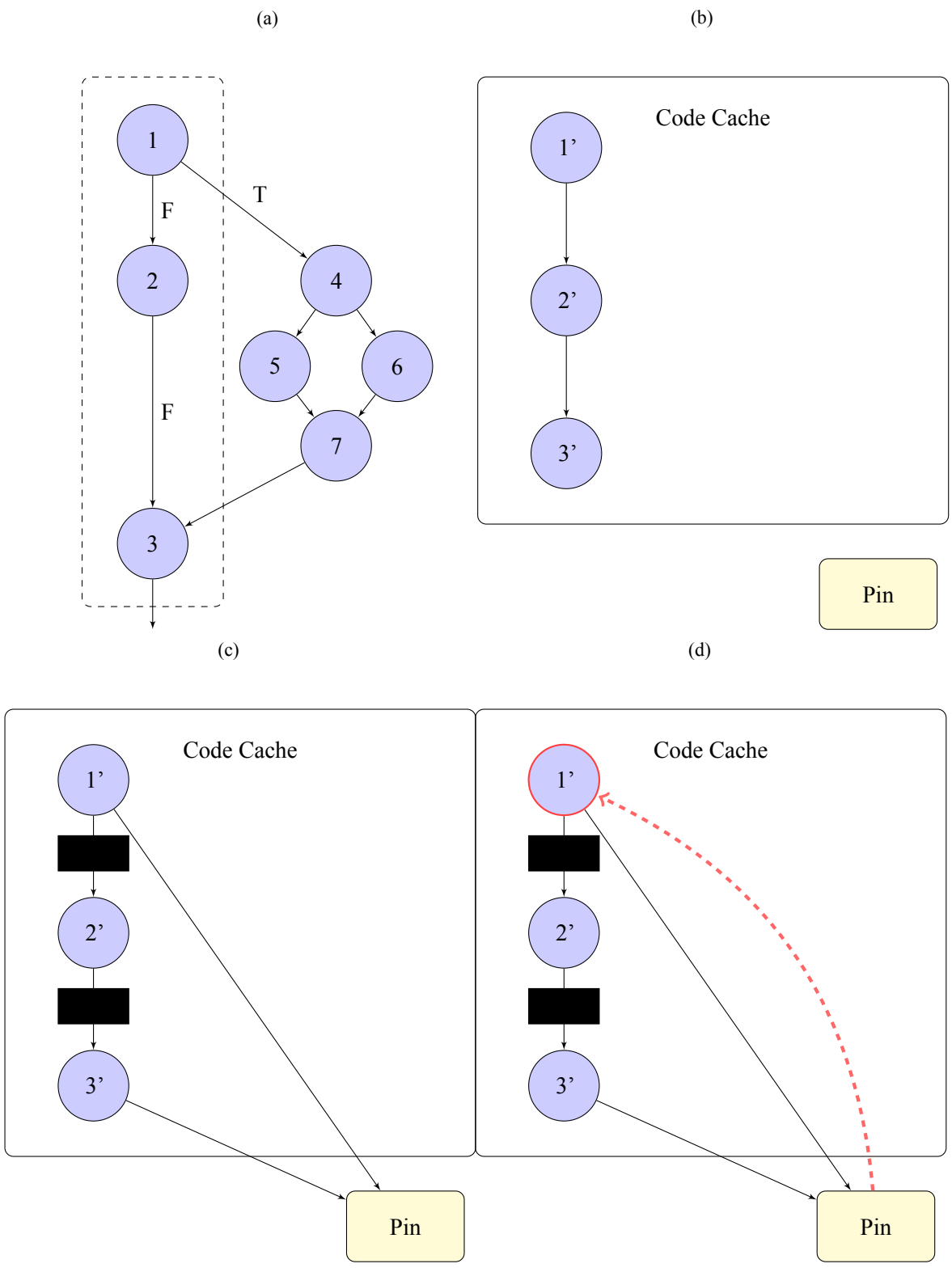



Figure 4.5. (a) Trace (dashed rectangle) is a sequence of continuous instructions, with one entry point. Basic blocks (circles) have one entry point and end at first control transfer instruction. (b) Copy of the working trace in the code cache. (c) Inserted Instrumentation and exits point to Pin. (d) Pin transfers execution to the trace in code cache.

Chapter 5

# Fault Injecting Tool

After describing how the framework that was used for our fault injection model works and what it provides, in this chapter our fault model will be presented. This work is about a fault injection tool that operates on binaries at run-time and on real hardware. It tries to emulate hardware transient faults by tampering, the specified by the user, assembly instructions. Needless to say, because of Intel's framework usage, our implemented tool runs only on Intel's Instruction Set Architectures (ISA).

## 5.1 Our Fault Injection Philosophy

As have been stated, there are various types of fault injection techniques. We wanted to find a way to inject faults to applications without having them to run on a simulator and thus the execution take a lot of time. So we implemented a tool that corrupts the application's *native execution* by injecting faults in the assembly instructions dynamically.

### 5.1.1 Injecting x86_64 Instruction Set Architecture

By native execution we mean that the application is being running on the processor, using of course the ISA of the processor. What we interfere with, is the ISA. This tool is implemented to inject faults on x86_64 and XEON assembly instructions. x86_64 ISA is the common one that Intel uses for the implementation of its processors. It follows a Complex Instruction Set Computing (CISC) processor design which means that a single instruction can deal with memory interference (load or store), register load or store, immediate values and complex memory references calculation. In contrast to CISC, in Reduced Instruction Set Computing (RISC) processor designs, there are different instructions (or opcodes) for

a memory load, a memory store and also different instructions for the various sizes of data that they manipulate.

In order to inject faults to instructions, we must first categorize them. In a RISC processor design it would be easy: The classification rules for fault injection would be imposed by the instruction names for the reason stated previously. So, if the RISC architecture was MIPS, then the class that has to do with memory loads would contain the instructions `lw`, `lh` and `lb`. The class that has to do with memory stores would contain the instructions `sw`, `sh` and `sb`. So, the application of a type of fault is straightforward on all the instructions of a single class. E.g. for the class with memory loads, someone would inject a fault in the address or the contents of the address that the load uses. As a result, when our tool encountered a `lw`, it would figure out that it is an instruction mapped to memory load instructions and the respective fault injecting routine would be called.

On the other hand, with a CISC processor design such as x86_64, we cannot use the instruction names in order to assign fault injecting routines to instructions. E.g. Instruction `add` has 5 different forms:

```
add <reg>, <reg> ; reg = reg + reg ⟶register load and store
add <reg>, <imm> ; reg = reg + imm ⟶register load and store, constant
add <reg>, <mem> ; reg = reg + *mem ⟶register load, store and memory load
add <mem>, <reg> ; *mem = *mem + reg ⟶register load, memory load and store
add <mem>, <imm> ; *mem = *mem + imm ⟶memory load and store, constant
```

So it goes with other type of instructions such as `mov`, `push` etc. The decision that we made is to assign a set of fault types to every instruction encountered and then choose one and change it according to the type. The fault types that we thought of are these cases:

- Memory load
- Memory store
- Register load
- Register store
- Jump

Each case is associated with a probability. Generally, this equation applies:

$$P_f(Mem\_store) \geq P_f(Mem\_load) > P_f(Reg\_store) \geq P_f(Reg\_load) \geq P_f(Jump)$$

where $P_f(X)$ is the probability that a fault is injected to case $X$.

After describing the way the instructions are classified, it is time to show the procedure to determine the fault type that an instruction will be associated with from a high level view. The below listing is a pseudocode that describes the fault injection mechanism without getting in details.

```
for every instruction:
   # 1. Identify what the instruction does from the cases
   # 2. Fill a set with all things the instruction does
   # 3. Choose simulating a darts game a fault case
   # 4. Return it
   fault_type = determine_fault_injection_class(instruction)

   # According to the fault, associate the instruction with
   # a fault injecting routine
   register_fault(instruction, fault_type)
```

For example, if we encounter an `add eax, ebx`, then the set of possible places to inject a fault is: Reg_load, Reg_store. The choice of the fault from the set is done by a random number generator on the discrete distribution that was built using weights for every case. The weights are the probabilities $P_f$ mentioned before. The whole process is similar to a *darts game*. The lowest probability-to-happen cases to happen are in the center of the target. As the probability grows, the cases move further from the center of the target. The random number generator ”throws” the dart at the target and in order to determine the fault case. After the fault case is chosen the corresponding routine will be called and the fault will be injected.

## 5.2 Implementation with Pin

### 5.2.1 Tool Configuration

The tool runs on a configuration file. This file must be provided by the user. In this file the user must specify the opcodes of the instructions to be injected with faults followed by the period of a single fault to happen for every instruction.

The syntax of the information in the configuration file is simple. The user specifies an instruction per line. The name of the instruction, which must be uppercase, is followed by the *period* of fault injection (in dynamic instructions) in this specific instruction and by a standard deviation (*std*) limiting possible *period* failures. Currently, the *std* is not used, because Pin handles precisely the frequency given and in the worst case the deviation is 1 to 2 instructions. Also, line comments are supported starting with a `#`.

A sample configuration file that injects floating point, integer arithmetic multiplication and division instructions is the one below:

```
# FP MUL-DIV
FDIV   1e2 0
FIDIV  1e2 0
FIMUL  1e2 0
FMUL   1e2 0
FMULP  1e2 0
```

```
#
# INT MUL-DIV
DIV    1e3 0
IDIV   1e3 0
IMUL   1e3 0
MUL    1e3 0
```

### 5.2.2 Tool Output

The tool as it runs creates and populates a file named `pinfault.out` that contains the report. The reporting consists of which instructions were instrumented with which fault type and the faults that were placed. Also, for each fault, there is a description of the steps that were followed to inject the fault. For example, the below listing shows a report for the injection of a register load fault in an instruction:

```
>> Enable register store fi.
 Place fault --> add eax, 0x1
curDynInst[ADD] = 1
 >< eax -> xmm5
 * save eax
 * add eax, 0x1
 * store to xmm5 instead of eax
 * restore saved result to eax
```

The details of register store fault injection and of the others will be described at the Type of faults subsection.

### 5.2.3 Instrumentation

It is time to describe how the whole process works with Pin instrumentation framework. The pseudo-code given in section 6.1 and describes how the faults are chosen happens in the instrumentation phase.

When the Pin's engine picks traces of code to instrument, for every instruction that is in the cache, `determine_fault_injection_class` is called to determine what kind of fault will be injected. After determining the fault, `register_fault` is called to register an *analysis routine* to the instruction according to the fault class. As have been said in the previous chapter, after the instrumentation of a trace finishes, the instructions will be executed and with them the analysis placed. For all the time that the instrumented instructions are in Pin's cache, they will be associated with the same analysis routines. If the cache replaces them, new instrumentation and thus new faults may apply through the analysis routines to these instructions. In the next section we will see which are these analysis routines and how they achieve their goals.

## 5.3 Type of faults

Everything happens in `register_fault` function. In this section we will walk through the type of faults that the fault injection tool supports.

### 5.3.1 Memory Load & Store

Memory operands in x86_64 assembly have this general form:

$$Imm(E_b, E_i, s)$$

where $Imm$ is a constant, $E_b$ the base register, $E_i$ the index register and $s$ a scaling factor which is 1, 2, 4, or 8. The final address in this form is calculated by this expression:

$$M[Imm, R[E_b] + R[E_i] * s]$$

where $M[x]$ the contents of memory address x and $R[x]$ the contents of register x. The most simple form of memory reference is the one that uses a constant. All the combinations can be seen at the table 5.1.

| | |
|---|---|
| $Imm$ | $M[Imm]$ |
| $(E_a)$ | $M[R[E_a]]$ |
| $Imm(E_b)$ | $M[Imm + R[E_b]]$ |
| $(E_b, E_i)$ | $M[R[E_b] + R[E_i]]$ |
| $Imm(E_b, E_i)$ | $M[Imm + R[E_b] + R[E_i]]$ |
| $(, E_i, s)$ | $M[R[E_i] * s]$ |
| $Imm(, E_i, s)$ | $M[Imm + R[E_i] * s]$ |
| $(E_b, E_i, s)$ | $M[R[E_b] + R[E_i] * s]$ |
| $Imm(E_b, E_i, s)$ | $M[Imm + R[E_b] + R[E_i] * s]$ |

Table 5.1: x86_64 instruction forms

This type of fault consists of two possible different way to inject a fault to the instruction: By tampering the address that the instruction uses or by tampering the contents of the memory that the address is associated. The choice between those possibilities is random.

For the first type of fault, as can be seen at table 5.1 the calculation of a memory address involves arithmetic operations, register loads and memory loads. In our implementation we do not analyze in detail how the final address is calculated, so we assumed that from the moment all these calculations happen, somewhere a fault would happen in the procedure. We do not know the type of the fault, but we decided to map it in such a way that the resulting address can be the previous address with a random bit-flip. The address of the transaction as an information is provided by the Pin framework.

The injection of the contents of a memory address is done provided that we have the address and the size of the transaction. All of these can be passed as arguments to the analysis routines during the instrumentation. Then, a random bit of the memory to be accessed is chosen and a bit-flip is performed.

This kind of injection has one limitation. Only memory load instructions can have their memory corrupted and that is because of a Pin's limitation. In fact, the memory load/store faults are injected through memory operand rewriting (a method referred in the previous chapter). It is technique that is done before the execution of the instruction. So when a memory load fault injection in the contents would be valid after the execution of the instruction, the changes of the contents of a memory store, would be overwritten by the assignment to this memory region.

Code names: `MEM_LD_CONTS`, `MEM_ST_CONTS`, `MEM_LD_ADDR`, `MEM_ST_ADDR`

### 5.3.2 Register Load/Store Contents

Instructions use values found in registers. A hardware malfunction could result in loading tampered values in them. This tool can inject a bit-flip in the contents of a register that an instruction uses. The register could be either a store or load register.

Code names: `REG_LD_CONTS`, `REG_ST_CONTS`

### 5.3.3 Replace Register Load

The Central Processing Unit (CPU), when executes an instruction that addresses a register, finds its value by looking at the *register file*. A possible fault that could happen is the loading of a value from a wrong register.

That said, at instrumentation phase a random register is chosen and its data are loaded, instead the correct ones that the instruction needs. For example, if the instruction was `add eax, 0x5` and `ebx` register was chosen, the value from `ebx` is stored in `eax` before the execution of the instruction:

```
add eax, 0x5

; becomes:

; analysis before instruction
PIN_GetContextRegval(current_context, ebx, ebx_val);
PIN_SetContextRegval(current_context, eax, ebx_val);

; instruction execution
add eax, 0x5

; analysis after instruction
PIN_ExecuteAt(current_context);
```

Code name: `REG_LD_RPLC`

### 5.3.4 Replace Register Store

Register store based faults are done in the same fashion as with register loads. Now, a possible fault that could happen is storing a value to a wrong register.

That said, at instrumentation phase a random register is chosen in which the store will take place. For example, if the instruction was `add eax, 0x5` and `ebx` register was chosen, the result will be stored in `ebx` instead of `eax` after the execution of the instruction. After its execution, the original value of `eax` is restored:

```
add eax, 0x5

; becomes:

; analysis before instruction
PIN_SaveContext(current_context, saved_context);

; instruction execution
add eax, 0x5

; analysis after instruction
PIN_GetContextRegval(current_context, eax, eax_val);
PIN_SetContextRegval(current_context, ebx, eax_val);
PIN_GetContextRegval(saved_context, eax, eax_val);
PIN_SetContextRegval(current_context, eax, eax_val);
PIN_ExecuteAt(current_context);
```

Code name: `REG_ST_RPLC`

### 5.3.5 Jump

Speaking of jumps we mean both only conditional. Unconditional jumps consist of a jump instruction and the address to jump. But the only way to inject a fault to the address is to do it at instrumentation stage. This will result in replacing a number of jump instructions that the mechanism would meet after such a fault has been chosen with new ones with another target address. Due to this limitation, the fault is not transient. It would be permanent.

On the other hand conditional jumps are formed by a pair of a comparing instruction and then a jump instruction, e.g. `jne <label>` that checks the result of the comparison. x86_64 assembly uses `cmp <op1>, <op2>` for comparisons and saves the result to the corresponding bit of the *status register* `eflags`. Then a jump instruction, e.g. `jle <label>` branches to `label` if the bit from the status register that has to do with the statement *less than or equal to* is 1. So all we have to do is to track the comparison instructions and if they are followed by an instruction that may branch, inject a fault in the appropriate bit(s) if status register right after the comparison is made. In this way we sabotage the decision that a

branching instruction makes.

Code name: `BRANCH`

## 5.4 Architectural pipeline mapping

In order to test the application in terms of validity, we should compare its results with a working tool. The tool that will be used for comparison is GemFI. So, the types of faults that are injected through this tool should match with the ones that GemFI injects.

GemFI injects faults in the pipeline. The fault types that it supports, are shown in table 5.2 at the first column. The second column is the list of the faults that this tool supports and which of them are equivalent to the ones that GemFI supports.

| GemFI | Pin-fault-injector |
|---|---|
| Decode | `REG_ST_RPLC`<br>`REG_LD_RPLC` |
| Memory Related | `MEM_ST_ADDR`<br>`REG_LD_RPLC` |
| Store/Load | `MEM_ST_CONTS`<br>`REG_ST_CONTS && MEM_LOAD_X` |
| Execution | `REG_ST_CONTS` |
| Branch | `BRANCH` |

Table 5.2: The types of faults GemFI injects and the corresponding ones that this tool injects.

With the sets of faults that are formed in the table 5.2, below there are the descriptions of each one fault types for the pipeline.

- *Instruction Decode:* Change the chosen register during load or store.
- *Branch Instructions:* Change the flow of the program injecting the conditionals
- *Memory Related Instructions:* The load/store address of the transaction changes.
- *Execution:* The contents of the store register are changed.
- *Load Instructions:* Change the contents of the register contents after the load.
- *Store Instructions:* Change the contents of the memory after the store.

Finally, all the fault types can result in three general ones that better depict fault on the pipeline:

- Decode
- Issue-Execute-Write back (IEW)
  - Branch Instructions

  - Memory Related Instructions
  - Execution

- Load/Store
  - Load Instructions
  - Store Instructions

# Chapter 6

## Experiments and results

Finally comes the stage of actually using the implemented tool. The first step is to evaluate its functionality. In order to do this, the same experiments will also be run on the GemFI [5] system and the outcomes will be compared. After this step, a series of executions with different configurations will be run in order to calculate the performance of the tool; the overhead of our tool to the original executable's time and the speed up against the simulation-based fault injection that GemFI uses. For these purposes, two well known operations will be tested. Sobel filter and Discrete Cosine Fourier (DCT).

## 6.1 Functional Validation - Micro-benchmarks

Before running the Sobel application and check how it behaves under this fault injection tool, it is mandatory to check each fault type that it inserts in terms of functionality. For this reason, the functional evaluation process consists of two steps: The first one tests the tool for every fault type that it supports on really simple programs. The second step has to do with running campaigns on the applications explained in the previous section and comparing the results with those of GemFI's.

### 6.1.1 Register store/Register load

In order to check the functionality on the register load/store fault injection, I used the `test1` function shown in figure 6.1. This snippet contains two `ADD` instructions (figure 6.2). They are of `<reg>, <imm>` type. So the fault types that can be injected are register load and store on `<reg>`. So I inform the tool via the configuration file to inject only `ADD` instructions.

For the register replacement fault type, after running some times the injection mechanism I observed that it indeed injects a fault in one of the two `ADD` instructions. And the injection of one affects the value

```
void test1(void)
{
    int i = 0;
    int j = i + 1; // j becomes 1
    int k = j + 1; // k becomes 2
}

void test2(void)
{
    int d = 66;
    d = 0;
    int i = d;
}

void test3(void)
{
    int i = 1;
    if (1 == i) {
        puts("1 equals 1");
    } else {
        puts("1 does not equal 1");
    }
}
```

Figure 6.1. Tests used to check the functionality of the fault injection mechanism. Full code can be found in Appendix A.

of the next one. For example, on a register load fault type, the register that contains `i` could be replaced from another register. So the resulting value on `j` could be something else. So, the value of `k` also is affected.

Table 6.1 shows the expected result of each variable and the ones that they got when they were injected.

| *var* | expected | REG_LOAD@1 | REG_STORE@1 | REG_STORE@2 | REG_LOAD@2 |
|---|---|---|---|---|---|
| *i* | 0 | 0 | 0 | 0 | 0 |
| *j* | 1 | 2 | 0 | 1 | 1 |
| *k* | 2 | 3 | 1 | 0 | -65535 |

Table 6.1: Replace registers fault injection: Results in all possible fault cases. The *@X* means that the fault was injected in the *X-th* add instruction.

In the column `REG_LOAD` fault means that the loading of a register that is involved in the related instruction will be done by another register. In our case , in the case `REG_LOAD@1`, which affects the first `ADD` instruction, the register that is chosen, contains the value 1. So `j` becomes 2 and `k` 3, since only one fault is injected. In the same fashion one can read the other fault type results in the table.

In the same fashion I tested the fault injection in the register contents. In table 6.2, in parentheses

```
<test1>:
  mov   -0xc(rbp), 0x0
  mov   eax, -0xc(rbp)
  add   eax, 0x1
  mov   -0x8(rbp), eax
  mov   eax, -0x8(rbp)
  add   eax, 0x1
  mov   -0x4(rbp), eax

<test2>:
  mov   -0x8(rbp), 0x42
  mov   -0x8(rbp), 0x0
  mov   eax, -0x8(rbp)
  mov   -0x4(rbp), eax

<test3>:
  mov   -0x4(rbp), 0x1
  cmp   -0x4(rbp), 0x1
  jne   else
  call  puts("1 equals 1")
  jmp   endif
else:
  call  puts("1 does not equal 1")
endif:
```

Figure 6.2. Assemby code of tests, used to check the functionality of the injection mechanism.

| *var* | expected | REG_LD@1 | REG_ST@1 | REG_ST@2 | REG_LD@2 |
|---|---|---|---|---|---|
| *i* | 0 | 0 | 0 | 0 | 0 |
| *j* | 1 | 65(6) | 129(7) | 1 | 1 |
| *k* | 2 | 66 | 130 | 0(0) | 129(7) |

Table 6.2: Register contents fault injection: Results in all possible fault cases. The *@X* means that the fault was injected in the *X-th* add instruction.

are shown the injected bits of the contents.

### 6.1.2 Memory store/Memory load

A memory load/store fault happens either on the address or the contents of this address. In order to check if the faults are inserted correctly I used the `test2` function shown in figure 6.1.

At first I checked the tampering of the addresses. I injected a fault on the second `MOV` instruction. Although it is a memory load fault, a memory store one would perform exactly the same actions and results.

In `test2` output (figure 6.3) we can see that `d` never gets assigned with the new value of 0. From the report, we can see that the second assignment is redirected to the new address. So variable `d` preserves

```
test2 output:
  **test2 ..... [FAIL]
  expected:
         0x7fffaa738ea8 d: 0
         0x7fffaa738eac i: 0
  result:
         0x7fffaa738ea8 d: 66
         0x7fffaa738eac i: 66

report:
    MEM_STORE
  inject_bit: 0
  old=0x7fffaa738ea8
  new=0x7fffaa738ea9
```

Figure 6.3. test2 output and report on memory address fault injection.

its old contents and `i` gets the contents of `d`.

Now, using the same test, I will present a case where the contents of the memory address get injected. In this test the second `MOV` instruction will be injected. The fault type is a memory store contents one.

The output of `test2` and report is shown in figure 6.4. The value of `d` is 0 and before it is assigned to `i`, the memory of `d` gets tampered and as a result, the new value of `d` and `i` is 128.

```
test2 output:
  **test2 ..... [FAIL]
  expected:
         0x7ffe78ab6a68 d: 0
         0x7ffe78ab6a6c i: 0
  result:
         0x7ffe78ab6a68 d: 128
         0x7ffe78ab6a6c i: 128

report:
    MEM_LOAD
  inject_bit: 7
  Contents:
  old=0x0
  new=0x80
```

Figure 6.4. test2 output and report on memory contents fault injection.

### 6.1.3 Jump

In order to test the branching fault injection, I used the `test3` function. This functions contains only one `CMP` instruction. This instruction is followed by a `jump` instruction which makes the pair to perform

a conditional branching instruction. The result of `CMP` lies in `flags` register. So the mechanism should inject conditional branches by injecting this register. Figure 6.5 shows the output of `test3` and report.

```
test3 output:
  **test3 ..... [FAIL]
  1 does not equal 1

report:
    Flag reg JMP
  Original rflags value: 0x246
  Changed rflags value: 0x206
```

Figure 6.5. test2 output and report on memory fault injection.

## 6.2 Functional Validation - Statistical

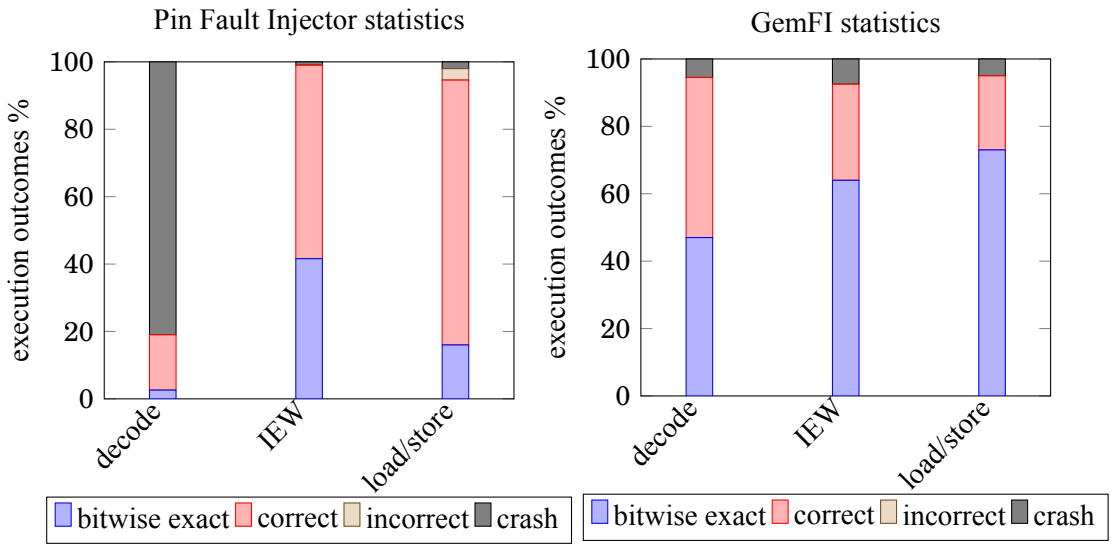
The statistical validation is the one that the results of this tool will be compared with these ones of GemFI. We expect that the statistics that the tools produce share common points.

Generally the possible outcomes of the execution after a fault injection on an application are the following:

- *Bitwise Exact:* The application terminates normally, and the end result is exactly the same, to the bit level, with that of a fault-free execution.
- *Correct:* The application terminates normally, and the end result (though not exactly correct) is useful or of acceptable quality.
- *Incorrect:* The application terminates normally, but the end result is useless or of unacceptable quality, based on some user/domain-specific metric.
- *Crash:* The application terminates abnormally, e.g., due to memory violation, division by zero, execution of an invalid instruction, etc.

As have been stated, the fault types that interest us are the Decode, IEW and Load/Store. The campaigns that took place on the Sobel application followed the steps described below:

1. Profile the dynamic instruction count of the application.
2. Produce a list of uniform numbers based on the instruction count. Each number is the number of instruction to inject.
3. Execute the fault injector for each number in the list and populate two output files. The first one is the injector's report for all executions. The second one is the applications result for all executions.




4. Merge the output files and produce the histograms. A produced histogram has information about all the fault types that the injector supports.

After these steps, I merged as described in the previous chapter the fault types and resulted with the statistics shown in the left graph of 6.2.

From the results we can observe a really big difference in the impact of faults injected in the Decode stage. This is somewhat expected because GemFI injects faults in the microcode that was produced by the original instruction, while Pin Fault Injector injects the instruction as is. So, in the case of GemFI, the granularity is finer and as a result the faults are less prone to cause errors.

For the other two fault types, we can see that the application generally produces correct results. With GemFI the most of them are exact, while with this tool are mostly correct as defined above.

## 6.3 Performance Evaluation

It is time to measure the overhead that our tool adds to the original execution time of each application. However, our tool injects faults. Faults that would not happen in the original application under normal conditions. So, we disabled the part of actually injecting the code and thus, only measure the rest of the mechanism described in the previous chapter.

### 6.3.1 FI tool Overhead

At first, I measured our baseline, which is the original tool-free execution time of the applications. This will be used in order to calculate the overhead that our tool introduces. Also, it is important to know which instructions and how many are in the main application function that we want to inject. Just to emphasize, all these sizes are dynamic and are dependent on the current input. A work could be done in order to see how these sizes scale for a larger input.

The workflow I followed for the measurements consisted of theses steps:

1. Determine instructions that the application uses.
2. Determine the count of each instruction found in 1.
3. Place them in tool's configuration file with along with the periods of injections.
4. Run the tool and get the results.
5. Enable/Disable instructions and/or update periods in the configuration and go to 4.

The 1. & 2. steps are done by using the pintool `insmix` that is included in the pintools folder that comes with the Pin kit. It has two options: Either emit all the instructions, along with their dynamic count, or categorized them by function. I used the second method in order to obtain instructions that have to do only with the core method of the application. E.g. the sobel's application core function is `sobel`, which performs the sobel filter on the input image.

In figure 6.6 the reader can see the impact of our tool with an empty configuration file. An empty configuration file, means that the tool will not have to search for any particular instruction to inject. As a result, the routines for determining the fault type are never called. So, we expect bigger slowdowns, when having a populated configuration file.

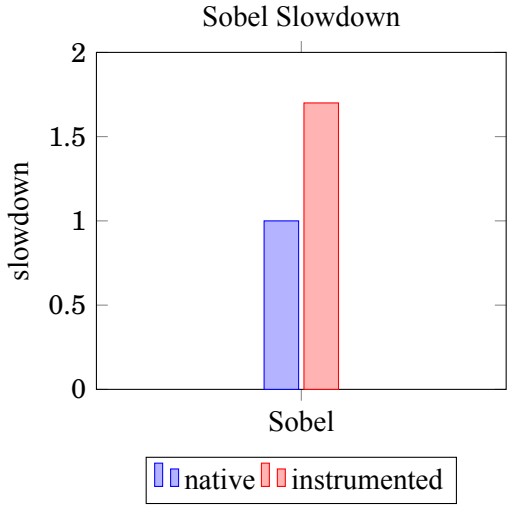



Figure 6.6: 1.7-1.8x slowdown with an empty configuration file

Figure 6.7, shows the slowdown that our tool introduces when having a populated configuration file. The instructions in this file are the ones that each application's core method consist of. Only instructions which have a great impact on the application are considered to be injected. Instructions of "great impact", are those which have a high frequency of execution and those that, although they execute rarely, are important for the result. Starting injecting instructions from higher frequencies and going lower, we

expect that the execution time of the application will descend. This expectation is verified as can be seen in figure 6.7.

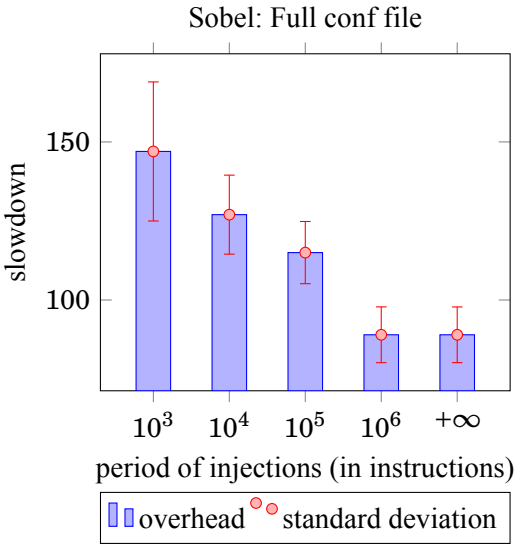



Figure 6.7: Application slowdown reduces as the period of injection per instruction increases.

As has been stated in a previous section, the majority of instruction in sobel application with the particular input are of the $10^6$ magnitude. So, if the period becomes bigger, no instruction will be executed. In order to show this the $\infty$ symbol is used. $\infty$ period means that the configuration file is populated, but a big period is selected in such a manner in which no faults will be injected. The observant reader will notice that the overhead remains the same even if no faults are injected. This phenomenon is absolutely normal, due to the fact that there are no big differences between injecting single digit number of faults and not injecting, while still searching for them.

## 6.4 Accuracy

All fault injection tools, should inject faults at specific points with a specific frequency. So it is important to insert all expected faults, or the most of them in order to consider the application accurate, from execution to execution. So:

> A fault injection mechanism that inserts all the expected faults is *accurate*. An *accurate* fault injection mechanism will insert faults with full accuracy and with no deviation from execution to execution.

As it has been stated, this tool injects faults given a period in instructions. In order to measure the robustness of this tool, we run the previous campaigns and calculated the accuracy by finding the *expected* and *placed* faults. It is generally expected, that while decreasing the period of injecting the accuracy should fall. But the deviation of the accuracy value should be nil.

Figure 6.8 shows that our tool is extremely accurate on the Sobel application. It is fully accurate and with no deviation.

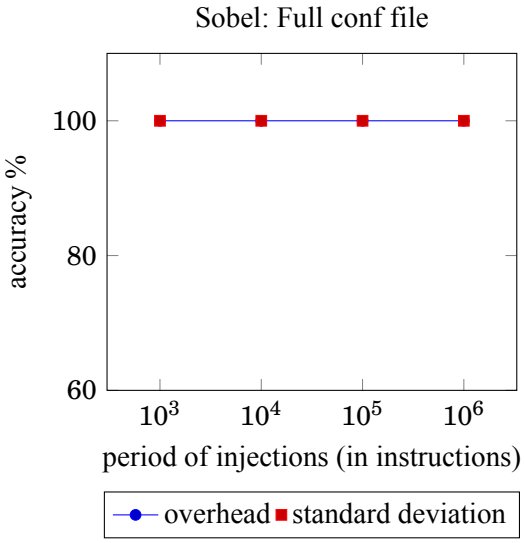



Figure 6.8: Accuracy of injections in Sobel application.

Chapter 7

# Conclusion

## 7.1 Summary

As we mentioned in the introduction the effort to improve the performance of computing system results in a new threat; The continuous shrinking of the transistors introduces many possibilities that the circuit is flawed, or prone to errors. As a result the reliability of the whole system is at question. There is research going to observe if there are applications that have low dependability under faulty states of the hardware, and also find mechanisms that would allow such applications to run over a faulty hardware. One tool that injects faults to application is GemFI and is based on simulator based fault injection.

The contribution of this dissertation is to provide another fault injection tool that injects faults in applications that run natively by tampering the assembly instructions at run-time. It is an approach that however introduces coarser grained fault injection compared to a simulator based tool, the campaigns would run faster. And that's because of the native execution. Also, there is no need to have the source of the application to inject. This thesis focuses on how this tool is built and finally shows that the results are close to the ones of the GemFI.

## 7.2 Future work

The next thing to do is to run more campaigns on different applications and compare the results with those of GemFI. Another important job is to determine the speed up of the time that campaigns take to finish, having as a baseline the time the GemFI takes.

Another considerable extension of this tool, is to make it inject also multithreaded applications. It then would be more like a tool for this era; the multicore kingdom.

Appendix

# Appendix A

```
#include <iostream>
#include "pin.h"

UINT64 icount = 0;

// Analysis routine
void docount() { icount++; }

// Instrumentation function
void Instruction(INS ins, void *v)
{
   INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

void Fini(INT32 code, void *v)
{
   std::cerr << "Count " << icount << endl;
}

int main(int argc, char * argv[])
{
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

Figure A.1. A pintool that counts dynamic instructions.

```
FILE *trace;

// Execution time routine:
// This function is called before every instruction and prints the IP.
VOID print_ip(VOID *ip) { fprintf(trace, "%p\n", ip); }

// Jitting time routine - Pin Callback:
// Called for every new instruction encountered.
VOID Instruction(INS ins, VOID *v)
{
   // Insert a function call to print_ip before the execution of the instruction
   // and pass the instruction pointer to it.
   INS_InsertCall(ins, IPOINT_BEFORE,
        (AFUNPTR)print_ip,
          IARG_INST_PTR,
        IARG_END)
}

int main(int argc, char *argv[])
{
   // Open file for tracing.
   trace = fopen("itrace.out", "w");

   // Initialize Pin.
   PIN_Init(argc, argv);

   // Add function that is called every time a new instruction is encountered.
   // This function is not called for instructions that are already in the code
   // cache.
   INS_AddInstrumentFunction(Instruction, 0);

   // Start the program. This call never returns.
   PIN_StartProgram();

   return 0;
}
```

Figure A.2. Example pintool that traces instruction addresses, taken from Pin's manual.

Appendix B

# Appendix B

The current appendix includes the tests that were performed in order to evaluate the *units* of the fault injection mechanism. With *units*, I refer to the various fault types that the mechanism supports.

First of all, comes the profiling of the target application. By profiling, I mean the number of dynamic number of instructions, with a given input. In order to extract those numbers, the first thing to do is to run an already existing pintool, `Insmix`. `Insmix` exists in the pin kit, along with other pintools, under the directory `source/tools/`. The execution is simple. If the under examination executable is called `tests`, then the profiling is done by issuing this command:

```
$PATH_TO_PIN_KIT/pin/pin -t
    $PATH_TO_PIN_KIT/pin/source/tools/Insmix/obj-intel64/insmix.so -r -- ./tests
```

The `-r` option is used to inform the tool to output information for each function separately.

After this step, I know the number of dynamically executed instructions for every function. So I take the information that I want and put them in the configuration file. The script that I used to execute the tests for each function is the one below:

```
#!/bin/bash

$PATH_TO_PIN_KIT/pin/pin -t $PATH_TO_PINTOOL/obj-intel64/pinfault.so -f $1 -c
    tests.conf -- ./tests
```

In order to run it, the user should provide as an command-line argument the name of the function that will be injected. In this case the arguments that I used were `test1`, `test2` and `test3`.

Finally, the source code, `tests.c`, of the tests taken is in the listing below:

```c
#include <stdio.h>

static void test1(void)
{
    int i = 0;
    int j = i + 1;
    int k = j + 1;

    printf("**test1 ..... ");
    if (k == 2) puts("[SUCCESS]");
    else puts("[FAIL]");
    puts("expected:");
    printf("\ti: %d\n", 0); printf("\tj: %d\n", 1); printf("\tk: %d\n", 2);
    puts("resulting:");
    printf("\ti: %d\n", i); printf("\tj: %d\n", j); printf("\tk: %d\n", k);
}

static void test2(void)
{
    int d = 66;
    d = 0;
    int i = d;

    printf("**test2 ..... ");
    if (i == 0) puts("[SUCCESS]");
    else puts("[FAIL]");
    puts("expected:");
    printf("\t%p d: %d\n", (void *)&d, 0); printf("\t%p i: %d\n", (void *)&i, 0);
    puts("resulting:");
    printf("\t%p d: %d\n", (void *)&d, d); printf("\t%p i: %d\n", (void *)&i, i);
}

static void test3(void)
{
    int i = 1;

    printf("**test3 ..... ");
    if (1 == i) {
        puts("[SUCCESS]");
        puts("1 equals 1");
    } else {
        puts("[FAIL]");
        puts("1 does not equal 1");
```

```
    }
}

int main(void)
{
    puts("--- TESTS -------");
    test1();
    putchar('\n');
    test2();
    putchar('\n');
    test3();

    return 0;
}
```

In order to compile you can use one of the below commands:

```
// If you have make installed
make tests CFLAGS=-O0
// or
gcc -O0 tests.c -o tests
```

# Bibliography


[1] Patrick P Gelsinger.
Microprocessors for the new millennium: Challenges, opportunities, and new frontiers.
In *Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE International*, pages 22–25. IEEE, 2001.

[2] Mark Seaborn and Thomas Dullien.
Exploiting the dram rowhammer bug to gain kernel privileges.
In *Black Hat conference, https://www. blackhat. com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Ker nel-Privileges. pdf*, 2015.

[3] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco, et al.
A survey on fault injection techniques.
*Int. Arab J. Inf. Technol.*, 1(2):171–186, 2004.

[4] Parag K Lala.
Transient and permanent fault injection in vhdl description of digital circuits.
2012.

[5] Konstantinos Parasyris, Georgios Tziantzoulis, Christos D Antonopoulos, and Nikolaos Bellas.
Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates.
In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 622–629. IEEE, 2014.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al.
The gem5 simulator.
*ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[7] Jiesheng Wei, Abu Thomas, Guanpeng Li, and Karthik Pattabiraman.
Quantifying the accuracy of high-level fault injection techniques for hardware faults.
In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 375–382. IEEE, 2014.

[8] Anna Thomas and Karthik Pattabiraman.

Llfi: An intermediate code level fault injector for soft computing applications.
In *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.

[9] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood.
Pin: building customized program analysis tools with dynamic instrumentation.
In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[10] Bryan Buck and Jeffrey K Hollingsworth.
An api for runtime code patching.
*International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[11] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al.
Dynamic instrumentation of production systems.
In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[12] Andrew Edwards, Hoi Vo, and Amitabh Srivastava.
Vulcan binary transformation in a distributed environment.
2001.

[13] Jonas Maebe, Michiel Ronsse, and Koen De Bosschere.
Diota: Dynamic instrumentation, optimization and transformation of applications.
In *Compendium of Workshops and Tutorials held in conjunction with PACT'02*. Citeseer, 2002.

[14] Derek L Bruening.
*Efficient, transparent, and comprehensive runtime code manipulation*.
PhD thesis, Massachusetts Institute of Technology, 2004.

[15] Kevin Scott, Naveen Kumar, Siva Velusamy, Bruce Childers, Jack Davidson, and Mary Lou Soffa.
Reconfigurable and retargetable software dynamic translation.
In *Proceedings of the 1st Conference on Code Generation and Optimization*, pages 36–47, 2003.

[16] Nicholas Nethercote and Julian Seward.
Valgrind: A program supervision framework.
*Electronic notes in theoretical computer science*, 89(2):44–66, 2003.

[17] Tevi Devor.
Pin: Intel's dynamic binary instrumentation engine - pin tutorial.
In *International Symposium on Code Generation and Optimization*, 2013.