Lecture Dependable Systems
Practical Report
Software Implemented Fault Injection

Paul Römer Frank Zschockelt

July 31, 2010

Contents

1 Introduction **3**

2 Software Stack **3**
- 2.1 The Host and the Virtual Machine . . . 3
- 2.2 The Fault Injector Kernel Module . . . 4
- 2.3 The Automated Fault Injection Framework . . . 4

3 x86 Registers **5**
- 3.1 General Purpose Registers . . . 5
- 3.2 Control Registers . . . 8
- 3.3 Segment Registers . . . 8
- 3.4 EFLAGS — Processor State Register . . . 8
- 3.5 Memory Management Registers . . . 10

4 The Experiment **10**
- 4.1 Evaluation . . . 10
  - 4.1.1 %ebx vs. %eXx . . . 11
  - 4.1.2 %esp . . . 11
  - 4.1.3 %edi,%esi . . . 11
  - 4.1.4 The Direction Flag . . . 12
  - 4.1.5 The Virtual x86 Mode . . . 12
  - 4.1.6 CR3 or MSB vs. LSB . . . 12

5 Conclusion **13**

6 Future Work **13**

1 Introduction

The topic of this seminar report is the analysis of injected single bit stuck-at and bit-flip faults into the Intel x86 processor registers under the fail-stop fault model by implementing a fault injector as Linux Kernel module. The report is the result of a project during the *Dependable Systems Lecture* at the *Hasso-Plattner-Institut* and describes the required software stack and components, the experiment and its results. It is motivated by the question how single bit faults harm the operating system and of which components this behaviour depends on.

2 Software Stack

The software stack consists of several parts that will be described in this section. It will give an overview of the used operating systems, the interaction between the virtual machine (VM) and the host, the kernel module itself and at least of the automated fault injection loop.

2.1 The Host and the Virtual Machine

As already mentioned the experiment uses a virtual machine[1] in which faults are injected. That has the following advantages over a hardware-hardware solution:

- easy communication between host and guest by using hosted pipes
- easy start, stop, restart and reset possibilities given by the VirtualBox command line interface
- snapshots that reset the virtual machine to a default state
- overall a faster development

But there is one important disadvantages: You don't know if the virtual machine affects the behaviour of the system after fault injection. For example you cannot be sure if the cpu abstraction layer of the VM influences the excuted code.

The Host uses

- OpenSuSE 11.2 as the running operating system,
- VirtualBox 3.0.6 by Oracle (earlier Sun) with 2 hosted pipes for communication reasons
- and a fault injection client written in Python.

[1] VirtualBox 3 www.virtualbox.org

The Virtual Machine uses

- Archlinux running the Linux Kernel in version 2.6.33 compiled with GCC 4.5,
- the system log redirected to /dev/ttyS0 to log kernel panics,
- a communication channel over /dev/ttyS1 for control commands,
- runlevel 3 with no ACPI but UDEV and HAL,
- and a fault injection server written in Python.

2.2 The Fault Injector Kernel Module

The fault injector module uses the timer interface of 2.6 kernels. It has the following parameters:

- interval: sets the timer interval
- inject_register: sets the register where the fault should be injected
- register_bit: chooses the bit of the register
- fault_type: can be one/zero/flip for stuck-at one/zero and bit flip faults

The module will inject the fault after every timer interval, imitating permanent processor faults with small intervals.

2.3 The Automated Fault Injection Framework

Because of the very high number of possibilities to manipulate x86-Register an automated fault injection was needed. Therefor a *FaultInjectionServer* (FIS) was written that controls the *FaultInjector Kernel Module* within the virtual machine. This server itself is controled by a fault injection client (FIC) that runs at the host machine. The *FaultControlClient* itself is managed by the main module *FaultControl*. Its main task is to load a list of errors, let the FIS (over the FIC) inject the fault and log the results. All needed communication runs over 2 serial connections which end up in hosted pipes on the host machine side. One is needed to catch kernel panics (handled by *SystemLog* module), the other to send control commands to the FIS.

The Faul Injection Loop 2

- *read next fault* – The *FaultControl* reads the next fault in the given list
- *reset vm* – Resets the virtual machine by using the VBox command line interface
- *start logging* – Starts the *SystemLog* in a seperate thread
- *inject fault* – Commands the FIS to inject the current fault
- *log results* – Logs which fault was injected and the VM's behaviour after the injection

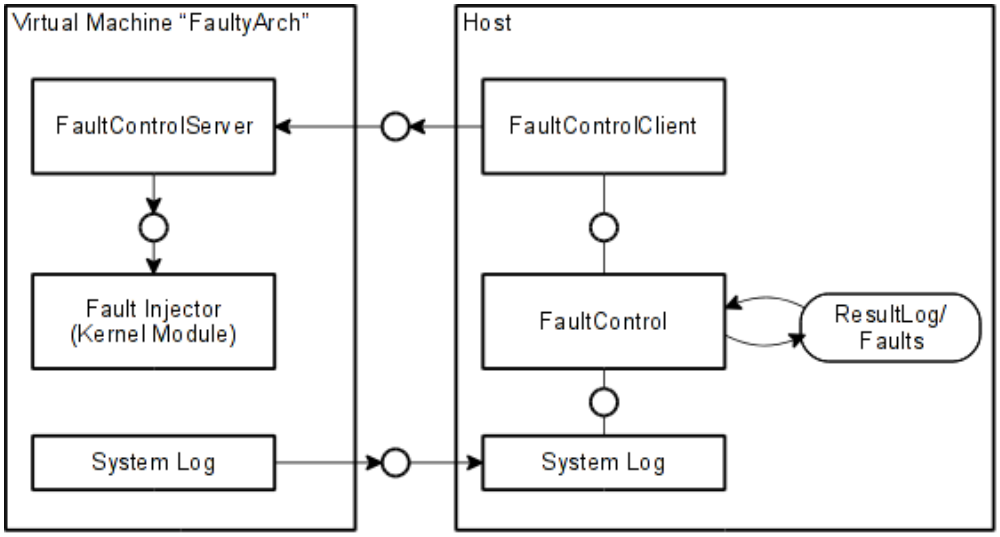

Figure 1: The complete setup in FMC

3 x86 Registers

The following section gives an overview about the most important x86 registers in 32 bit mode. These consists of the general purpose registers, control registers, segment registers, the processor state register EFLAGS and memory management registers. Each register group and each register's functionality will be explained next.

3.1 General Purpose Registers

These registers can store both data and addresses. They are the most often used registers for several task within programs. The prefix 'E' determines a wide of 32 bit.

EAX — The Accumulator Register is specialized for calculations. Although most calculations can occure between any two registers, there are special instructions giving the accumulator special preferences.

EBX — The Base Register was a special address register in former (and 16 bit) days. Today it is a register which can be used to store everything.

ECX — The Counter Register is in higher programming languages often know as 'i'. Several looping instructions decreases this counter while operating on data.

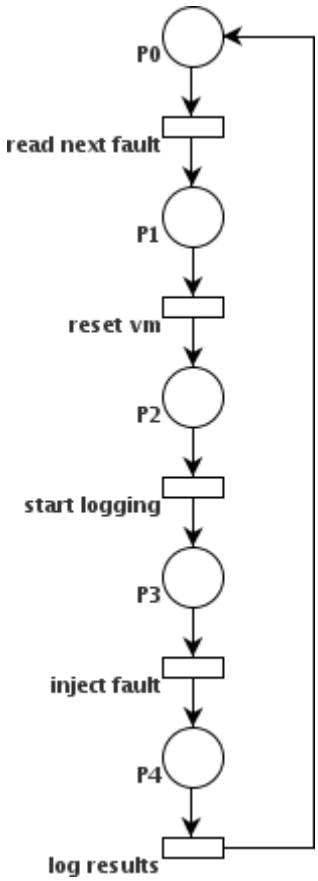

Figure 2: Fault Injection Loop

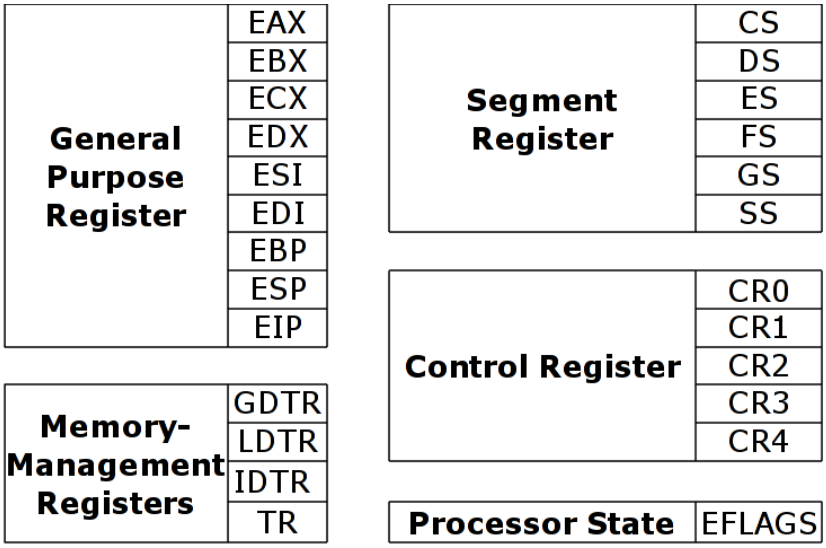

Figure 3: X86 registers overview

EDX — The Data Register is most coupled to the accumulator. Instructions dealing with oversized data using this register to store the most significant bit. It is said that for calculations only the accumulator and the data register are needed.

EDI — The Destination Index is needed whenever an instruction creates data which has to be stored in memory. Therefor it contains the address to the destination in memory. In conjunction with the source index it is used to read and store data by using special instructions which automatically increase or decrease the indices.

ESI — The Source Index is, as the already explained destination index, used for data reading and storing.

EBP — The Base Pointer is part of the x86 function-call mechanism. Whenever a function is called the base pointer is set to the stack pointer. Within the function block only the base pointer will be used to access parameters and to refer to variables.

ESP — The Stack Pointer is the second part of the x86 function-call mechanis. It handles function calls and later on gives up its address to the base pointer.

EIP — The Instruction Pointer holds the address of the next instruction to call.

3.2 Control Registers

These registers change or control the general behaviour of the processor. There are five registers defined in the x86 specification but only four can be manipulated because register CR1 is reserved. As the general purpose registers they are 32 bit wide.

CR0 consists of various control flags which change the general processor behaviour.

CR1 is the reserved register[2].

CR2 is needed for handling page faults. That means after a page fault the address leading to the fault is saved in this register.

CR3 is also needed for paging. That means this register contains the Page Directory Base Register to look up page tables for the current task.

CR4 is an extension to CR0 and only used in protected mode. It also consists of various control flags that change the processor's behaviour.

3.3 Segment Registers

The Intel x86 processor architecture memory addresses are devided into two parts: segments and offsets. The segment specifies the beginning of the allocated memory block, the offset the index into it.

CS — The Code Segment contains the segment of the current instruction (IP as offset).

DS — The Data Segment is used by several data instructions.

ES, FS, GS — The Extra Segments are used by various instructions.

SS — The Stack Segment contains the segment of the current stack (SP as offset).

3.4 EFLAGS — Processor State Register

The 32 bit wide EFLAGS register contains the current state of the processor. Table 4 gives a short description to each flag.

[2]Reading and writing seem to have no effects

0	CF	Carry Flag	Indicates when an arithmetic carry or borrow has been generated
2	PF	Parity Flag	Indicates if the number of set bits is odd or even in the binary representation of the result
4	AF	Adjust Flag	Indicate when an arithmetic carry or borrow has been generated out of the 4 least significant bits
6	ZF	Zero Flag	The result of an instruction was zero
7	SF	Sign Flag	Result of last mathematic operation resulted in a value whose most significant bit was set
8	TF	Trap Flag	The x86 processor will execute only one instruction at a time and then call interrupt 1
9	IF	Interrupt Enable Flag	Determines whether or not the CPU will handle maskable hardware interrupts
10	DF	Direction Flag	Autoincrement the source index and destination index (like movs) will increase both of them
11	OF	Overflow Flag	The overflow flag is set when the Most Significant Bit (MSB) is set or cleared
12,13	IOPL	I/O Priviledged Level	Shows the I/O privilege level of the current program or task
14	NT	Nested Task Flag	Controls chaining of interrupted and called tasks
16	RF	Resume Flag	Temporarily disables debug exceptions
17	VM	Virtual 8086 Mode Flag	Indicates that the task is executing an 8086 program
18	AC	Alignment Check Flag	Enable/Disable Alignment check
19	VIF	Virtual Interrupt Flag	Marks a pending interrupt
20	VIP	Virtual Interrupt Pending Flag	
21	ID	ID Flag	Supports the CPUID command

Figure 4: EFLAGS Register Overview

3.5 Memory Management Registers

The memory management registers mainly supports the interaction with the main memory. That means they are used for paging, correct interrupt handling and task switches.

GDTR — Global Descriptor Table Register points to the global descriptor table that defines the characteristics of the processors segment registers. It contains local descriptor tables.

LDTR — Local Descriptor Table Register points to the local descriptor table that defines the characteristics of local memory segment.

IDTR — Interrupt Descriptor Table Register points to the interrupt descriptor table that defines the correct response to interrupts and exceptions.

TR — Task Register contains a pointer to the task state segment of the current process.

4 The Experiment

The starting point of the experiment was the creation of a list of registers to provide a way to make suggestions. This list consists of the registers itself, a classification and a short description. For the sake of simplicity the injected fault was a single bit-flip only. The participant's task was to decide how the system reacts when the given register is manipulated. Therefor a "system harm level" was created:

1. No effect visible
2. Effects visible in the system log
3. Kernel Panic
4. System freeze

Meanwhile the suggestions were collected the automated fault injection created the real results. On base of these results and the made suggestions a list was created which contains all registers that led to an unexpected system behaviour (see 5).

4.1 Evaluation

This sections steps through the list of chosen suggestions and tries to explain the cause of the unexpected behaviour.

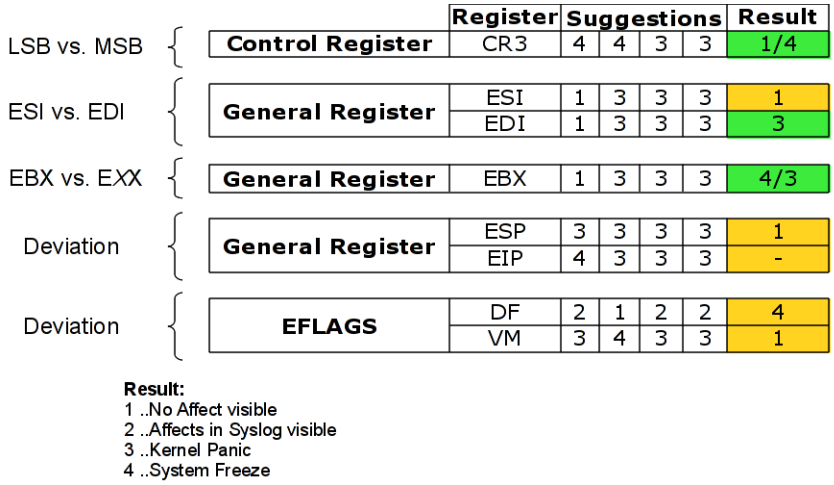

Figure 5: Chosen Suggestions

4.1.1 %ebx vs. %eXx

Interestingly only the manipulation of the ebx register lead to a failure. This phenomenom can be explained with the function calling conventions used by the gcc. It assumes that the values of some registers[3], including %ebx, should not be changed by a called function.

4.1.2 %esp

The results suggest that changing the stack pointer does not have any effects on the execution of the kernel. This was somehow unexpected, since it's basically used everywhere - for local variables, function parameters etc. It was clear that the fault injection has to be faulty. After looking at the assembler output of the injection function it became clear. gcc saved the stack pointer before executing code from the function to %ebp and copied it back at the end of the function. This is due to the compiler flag "-fno-omit-frame-pointer" which was used for compiling the kernel for debugging purposes.

4.1.3 %edi,%esi

It was expected that it would not make a difference whether %edi or %esi was changed, since both are used together for copying memory efficiently. So, either the processor would read or write from memory which should not be touched. But only %edi lead to a fault. Once again the compiler lead to this behaviour. If gcc does not need to copy

[3] `http://www.delorie.com/djgpp/doc/ug/asm/calling.html`

Control Register	CR3	4	4	3	3	1/4

Figure 6: CR3 Suggestion

memory around it will use those registers as general purpose registers. In the tested kernel %esi was not used in the call stack of the injection method, so injecting faults into %esi could not lead to a failure.

4.1.4 The Direction Flag

To understand why manipulating the direction flag leads to a system freeze you have to understand the behaviour of %edi and %esi first (see 3). The direction flag allows programs to modify the "direction" in which the registers change. That means setting the direction flag to '1' will lead to an automatic decrease, setting it to '0' will lead to an automatic increase of both registers when using special instructions. That means the manipulation of the direction flag leads to unknown behaviour because the instruction copies data from wrong addresses which in most cases leads to a system freeze in kernel mode.

4.1.5 The Virtual x86 Mode

The problem of setting the virtual x86 mode is that it does not work as easy as expected. That means manipulating the flag will not lead to a different behaviour because of security reasons. That is why single bit faults of this register will not harm your process. Nevertheless there are two ways to enter the virtual x86 mode:

- Manipulate the Task State Segment of another x86 task and load it.
- Manipulate the EFLAGS register (which contains the VM register) on stack and reload it by using an interupt return (ITER).

4.1.6 CR3 or MSB vs. LSB

As seen in 6 the suggestions were correct for the most significant bits of the control register: Manipulating the page directory base leads to a system freeze. But there is one important aspect: Changing the least significant bits (in fact bits 0 to 12) does not influence the system. In normal 32bit mode the bits 0 to 12 are not used and assumed to be 0 because the additional address space is not needed. However if you are using the physical address extension (PAE) feature the whole 32 bit are used and manipulations of the least significant bits will then influence the system's behaviour too.

5 Conclusion

It should be clear that simulating bit errors in the registers of an x86 will lead to crashes. But it was very interesting which registers are more susceptible to faults. Moreover it is not the register or the architecture of the processor which will define how the failure will look like, but the compiler. The implementation of the compiler defines how registers are used and when a register can be expected to have the same value as before a function call.

6 Future Work

The fault injection module has a major drawback. It only changes the register in the context of a kernel timer interrupt. This means that the injected fault will only lead to a failure if the faulty register is used somewhere in the call stack of the injecting function. This could be changed by implementing a kernel module which injects those faults into the task states, similar to a ptrace()-based approach by Volkmar Sieh. By this approach the location where a fault can be injected will be nearly random and could be expanded to user mode processes.