CORE

Metadata, citation and similar papers at core.ac.uk

Provided by Archive Ouverte en Sciences de l'Information et de la Communication

Fault-Injection using Virtualization for Critical Software Validation in Automotive

Antoine Blin, Youssef Laarouchi, Philippe Quéré

▶ To cite this version:

Antoine Blin, Youssef Laarouchi, Philippe Quéré. Fault-Injection using Virtualization for Critical Software Validation in Automotive. Embedded Real Time Software and Systems (ERTS2014), Feb 2014, Toulouse, France. hal-02272451

HAL Id: hal-02272451

https://hal.archives-ouvertes.fr/hal-02272451

Submitted on 27 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Fault-Injection using Virtualization for Critical Software Validation in Automotive

Antoine Blin[(1)], Youssef Laarouchi[(2)], Philippe Quéré[(2)]

[(1)]Contact author, Renault, antoine.blin@renault.com
[(2)]Renault

Abstract:

In the field of critical and embedded systems in the automotive industry, where human life is at stake, manufacturers and suppliers must develop robust and reliable systems while being confronted with real time, cost and energy consumption constraints. The increasing complexity of electronic and electronic control units (ECU) in the context of the automotive industry has positioned dependability in the heart of the concerns of manufacturers.

The ISO 26262 safety standard published in November 2011 in the automotive industry refers to fault injection during software integration and testing (part 6, chapter 10, requirement 10.4.3, table 13, method 1c). It is recommended for ASIL A and B, and highly recommended for ASIL C and D. The meaning is that for ASIL C and D a rational shall be provided if the injection tests are not performed.

In this paper, we first introduce the context concerning the fault injection technics, the software architecture in automotive, and the virtualisation technics. A state of the art and a quick discussion concerning each topic is presented to justify the choice we made in our work. In a second part we describe our proposal to perform fault injection using emulation on an x86 standard platform. In a third part we describe fault injection using virtualization with respect to specific embedded platform characteristics. We conclude on pros and cons of our proposals, by specifying the technical issues that were treated in each phase of our experimentations. We then present some prospective views for future work.

Keywords:

Fault injection, dependability, virtualisation, automotive, safety, ISO 26262

1 Introduction

Historically, the automotive software architectures were heterogeneous. To counter this situation, two new standards have been developed, AUTOSAR software architecture for real-time systems and GENIVI a Linux based open source platform for In-Vehicle Infotainment. These new automotive architectures aims at making as standard as possible the software interfaces between different software products. For instance, AUTOSAR defines the different software layers (Basic Software, AUTOSAR Runtime Environment, Software Components …) as well as the interaction between these layers. This standardization aims essentially at reducing cost of software products, which is a great challenge for car manufacturer.

In addition, these architectures have been deployed in parallel with the emergence of ISO 26262, a new safety standard dealing with embedded EE systems in automotive. This standard adds technical requirements on software that shall be respected during the whole lifecycle (design, development, validation…). In this paper, we are interested in specific ISO 26262 requirements; those dealing with fault injection. We assume that fault injection, a traditionally costly validation technique, can be done through a “low cost” approach when using virtualization. In next section, we briefly present the fault injection techniques, then the virtualization to inject faults.

2 Fault injection techniques

Fault injection is a technique for improving the robustness of the system by introducing faults to test error handling mechanisms. Different faults can be triggered according to the failure we are targeting to tolerate. As presented in [1], we distinguish between hardware faults and software faults. Each fault category can be triggered by different fault injection technique:

- Hardware Fault Injection is achieved through electrical or physical component deterioration. This fault injection type is particularly long and expensive. In fact, it requires the establishment of dedicated infrastructure and does not allow precise control of the location of injected faults.
- Software Fault Injection is achieved on target software in its environment. The injection can be done on the component itself or on its entries by altering the functioning through spatial (essentially memory space) or temporal (CPU, resource usage, etc.) unwanted interactions. This type of injection is generally presented as cheaper than the hardware injection since it does not require any special equipment. Moreover, the injection can be well mastered if the target of test is clearly specified as discussed later.

The fault injection common architecture can be summarized in the following points (cf. Figure 1):

The Library: contains all errors to be injected into the system. Its content is defined during the system design and especially when defining safety mechanisms. These mechanisms will be the target of test later in the process of fault injection.

1. The injector: designed to activate faults on the target system with respect to the safety mechanisms requirements previously included in the library.
2. The target is the system under test.
3. The monitor observes the injections and logs obtained results
4. The collector saves the data issued from the system.
5. The analyzer processes the collected data to determine whether the system has properly reacted to the injected faults.
6. The controller coordinates the fault injection with the system response observation process..

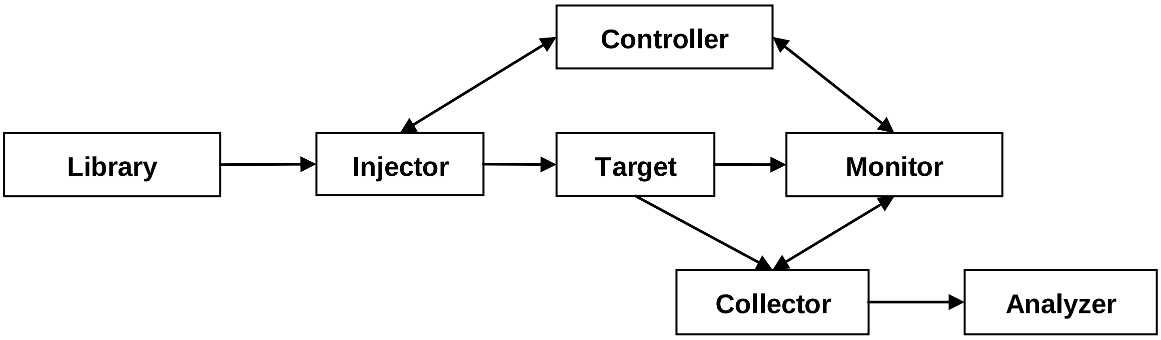


Fig. 1. Fault Injection Common Architecture

According to the test target, the Software Fault Injection can be applied, either on the operating system or on the application or on both. The choice of the injection layer (Operating System or Application) depends on the aim of the analysis. In our case, the safety analysis deals with an ECU functional behavior implemented in the applicative layer. We assume that the operating system is trusted and we focus on software safety mechanisms implemented in the application layer. Consequently, our targeted test system is the application implementing safety mechanisms and dealing with critical signals.

Notice that fault injection on application layer requires a strong interaction with the underlying Operating System. In fact, the injector module presented in Figure 1 can be implemented in different software layers:

1. The first layer is the application itself. In this case, we need to have access to the source code in order to insert the necessary mutations for fault injections. Notice that this implementation is too intrusive and requires the modification of the application source code, which is not always possible.
2. The second layer is a middleware which consists in altering the tested applications I/Os and tests its behavior. This implementation is less intrusive but does not affect the internal application behavior (for instance it does not inject fault in critical application functions).
3. The third layer is the Operating System. In this case, we implement an injector module to insert in the OS. This implies the modification of the OS which is not always possible, especially in industrial context.

We think that intrusive fault injections are the major inconvenient of classical Software fault injection. In this paper, we aim at performing non-intrusive faults injections on Software Applications. Layers 1 and 3 previously described are clearly intrusive and do not correspond to our objective. Layer 2 (middleware) is interesting but does not have sufficient coverage for the tested application.
We propose the use of an additional software layer between Hardware and Operating System. Such a layer would allow us alter OS I/O without altering the source code of neither the OS nor the application. This layer is technically enabled through the use of virtualization, as presented in next section.

3 Virtualization and Emulation

3.1 What is Virtualization?

Virtualization technology allows multiple operating system instances run simultaneously on the same hardware. Each virtualized OS (Called “guest OS”) is hosted in a logical container called “partition”. Different partitions may exist and run while being managed by a hypervisor (Called “host”).

The main hypervisor role is to share hardware resources between different hosted OS. Virtualization technology must guarantees strict isolation between partitions: if one virtualized system fails, the hypervisor ensures that it does not disturb other virtualized OS. The isolation mechanisms are important since they forbid spatial and temporal interference between hosted OS. Such properties are being considered closely in current R&D automotive industry. In fact, through virtualization, we would be able to make AUTOSAR real time system coexist with GENIVI on the same hardware.

3.2 Emulation Technique

An emulator is a program that emulates Operating Systems compiled for architecture (ARM...) on different machine architecture (X86, PowerPC, etc.). This mechanism is different from the virtualization in which the instructions of the host Operating Systems are directly executed on the target.

With emulation, the instructions of the guest Operating Systems are translated by the emulator into host architecture instructions. To emulate real time characteristics, the emulated clock rate is clocked with the number of emulated instructions executed. Therefore a real time system emulated cannot communicate with non-emulated devices plugged to the host.

In our work, we used both emulation and virtualization. Emulation was used when testing our fault injection approach on a standard x86 machine. During this phase, we explored the technique of injection from the emulator platform as detailed in section 4. Once the fault injection correctly set up, we worked on a real embedded target platform by using virtualization, as detailed in section 5.

4 Fault Injection on X86 Platform

Let us remind first our main objective: we aim at injecting faults on automotive application software, running on a real time automotive OS. In our work, the final hardware target was not easily available (delivery delay, cost, etc.). We decided to begin our study by using a standard x86 machine by emulating an embedded platform.

For this, we require the following main components:

- An automotive operative system respecting some hard real time requirements. This requirement aims at making the tested runtime environment as representative as possible of a standard automotive ECU platform. In our work, the chosen Operating System was Trampoline [9], an Open Source OS respecting AUTOSAR standard and running standard Basic Software.
- An emulator used to inject fault on basic software. In our case, QEMU [16] was the emulator of an i.MX27 ARM platform.

The fault injection architecture consists of six layers (cf. Figure 2):

1. The X86 Hardware
2. An open source Operative System: Linux
3. An emulator, QEMU, running as a standard Linux user mode process
4. The embedded emulated hardware (Armadeus apf27 board [10])
5. Trampoline: the embedded real time OS
6. The application software targeted by fault injection

As presented in section 2, fault injection can be done on different layers, and we are interested in the application layer, by injecting fault in the emulated (or virtualized) hardware. Many software injection fault types exist, depending on the objective and limitations of the tests. As we set as an objective to leave unmodified the source code of tested applications, the fault injection has to deal with the dynamic behaviour of the application. In this paper, we are interested in two software fault injection types that directly impact the application functioning:

- Fault injection on global variables: we modify application global variables to introduce non-consistent states.
- Fault injection on function calls: we modify functions parameters to impact program behavior.

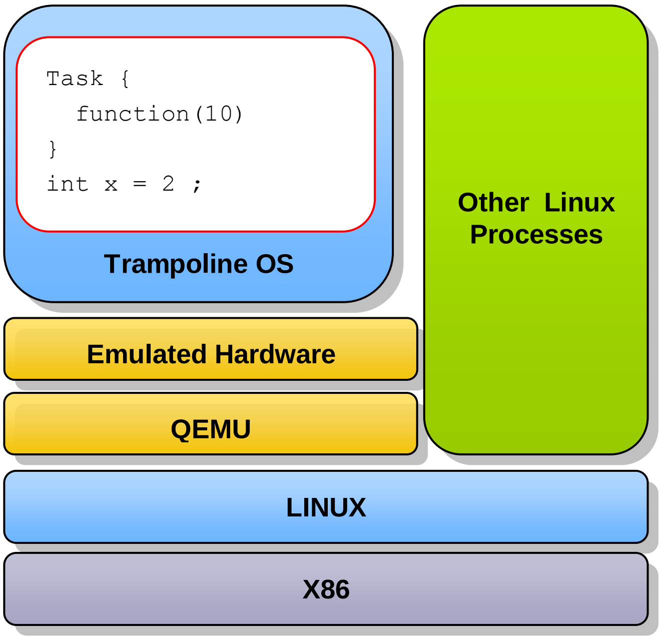


Fig. 2. Fault Injection Architecture

For both injection types, we suppose that we have a least access to the name of critical variables (generally used) and the critical functions signature (especially name and arguments). We strongly think that this hypothesis is realistic. In fact, from a car manufacturer perspective, it is always possible to have this information without being too intrusive in software implementation.

Notice that other software fault injection types can be considered (such as injections on control structure, local variables, etc.). However each introduced type needs to be linked to the application functioning first and then attached to the emulator and/or hypervisor structure to ensure correct fault injection.

4.1 Fault Injection on Global Variables

To fully understand the fault injection process we propose to first detail the different memory management strategies used by the different software layers.

Linux uses virtual memory management technic. Each process uses a virtual address space. Virtual addresses to physical addresses translation is done by the Memory Management Unit.

Trampoline doesn't use Memory Management Unit. Each address used in basic software is directly mapped to physical memory.

QEMU allocates physical emulated memory used by Trampoline on virtual memory. Furthermore, it implements a Software translation mechanism to translate physical emulated addresses to virtual addresses.

The fault injection process consists in the following steps:

1. Reading the physical emulated address of the targeted global
2. Converting it to the emulator virtual address
3. Updating memory pointed by emulator virtual address in the Linux page table
4. The MMU used by Linux converts virtual to physical addresses and update physical memory.

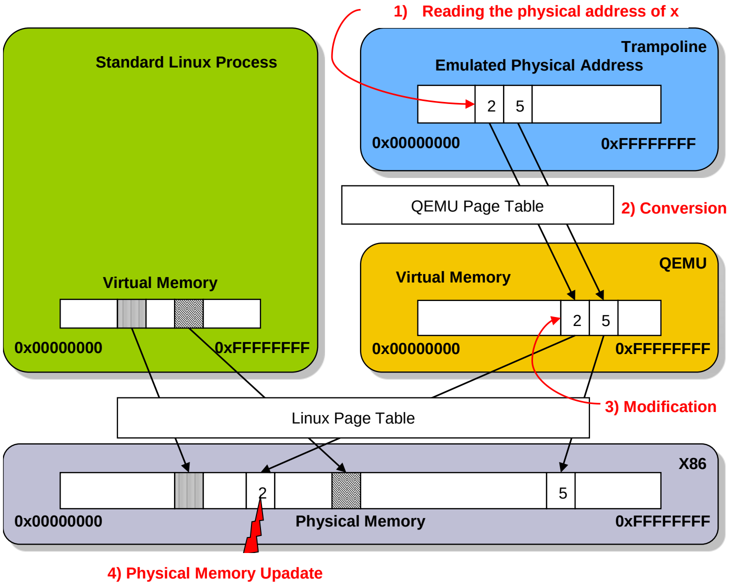


Fig. 3. Fault injection on Global Variables

The first step requires global variables (corresponding to a critical signal) address reading in the compiled binary (called “elf” file). The "elf" operating system format contains a header describing the general structure of the file and its sections. One of the sections presents the symbol table in which a translation between the names of functions and global variables on one hand and memory addresses on the other hand is made. Accessing and reading this table is a first important step in our fault injection process.

The next steps are set up on the memory management in Trampoline and QEMU.

For Trampoline, the system code and application is compiled into a binary, with no use of virtual memory device. The addresses used in the OS thus correspond to physical hardware addresses. This feature significantly simplifies the fault injection process. In fact, it is not necessary to translate application addresses into emulated addresses. Moreover, since no “swap” mechanism is used in Trampoline, it is not necessary to consider the case where OS memory pages have been temporarily saved on the emulated memory space.

For QEMU, the memory management QEMU uses two mechanisms:

- The mechanism of paged virtual memory, provided by Linux. Indeed, QEMU runs as a user process, its allocated memory (used for QEMU and emulated OS) uses standard Linux virtual addresses. At the start of emulation, QEMU allocates as much memory as needed by the emulated OS.
- A software translation mechanism aiming at translating each physical address required by the emulated OS into virtual address within the memory blocks allocated by QEMU (to the emulated OS). This mechanism uses an internal QEMU page table.

The fault injection is then done by a QEMU developed module. This module has as input the name of the variable to alter and the new value to affect and modifies directly the QEMU virtual memory as explained in Figure 3.

4.2 Fault Injection on Function Calls

Unlike fault injection on global variable in which the memory address of the variable remains unchanged, the function parameters are temporarily stored in the stack or in registers when the function is being called. We have to act when the function is executed to inject the faults on its arguments.

Our new fault injection process is updated as described below:

1. Reading the physical emulated address of the targeted function
2. Converting it to emulator virtual address
3. Setting a software breakpoint on this emulator virtual address
4. When breakpoint is triggered
5. The emulated OS is stopped
6. The arguments values is changed on the emulated stack
7. We restart the emulated OS with its new context

To implement this fault injection on function arguments within QEMU, it is necessary to first determine how the arguments are passed to functions called. These conventions depend on the hardware architecture for which the code is compiled, the compiler and the call conventions used. Conventions used by Trampoline are detailed on the official website of ARM Holdings and are quite complex. They can be summarized as following: the first four arguments are passed in registers; the following are passed on the stack.

This injection technic suffers from some limitations. For instance, when compiler optimizations (*inlining*) are enabled, function calls are integrated to the code and fault injection becomes impossible.

More generally, fault injection on emulated application has several limitations and benefits. One major benefit of emulation is the entire control of the timing aspects of emulated OS. In fact, the internal clocks of emulated OS are cadenced by the emulator; the emulator can freeze guest OS without temporal impact on its behavior. This makes fault injection overhead less. Moreover, emulation allows quick unit testing without deploying the software application on real target which is energy and time consuming. Limitations can be summarized in the following; emulation cost (all the hardware has to be entirely emulated), representativeness (when emulating the whole runtime environment).

5 Fault injection on embedded hypervisor

As previously discussed, emulation has several limitations and especially representativeness. In fact, emulation hides the complexity of runtime environments and makes some assumptions about the interaction of the application with its environment. The use of hypervisor is more representative of embedded platform. We propose to first present fault injection architecture and then detail the fault injection on global variables and functions.

5.1 Architecture

It‘s necessary to have three tools to perform a fault injection campaign on embedded platforms:

- an operating system running applications,
- a hypervisor used to inject fault on basic software
- the targeted platform board.

In our case, we used an open source hypervisor called *CodeZero*. This choice is justified by the fact that we need a hypervisor supporting ARM architecture, and *CodeZero* was one of the rare open source hypervisors fulfilling this requirement. Using *CodeZero* raised some restrictions. For instance, there were no OS running on this hypervisor, but just POSIX applications running in bare *metal mode*. Since we are interested in checking safety mechanisms in application, we performed fault injection on virtualized POSIX application (Figure 4). The platform used to launch wasPB926-Versatile (This board wasn't available we used QEMU to emulate it).

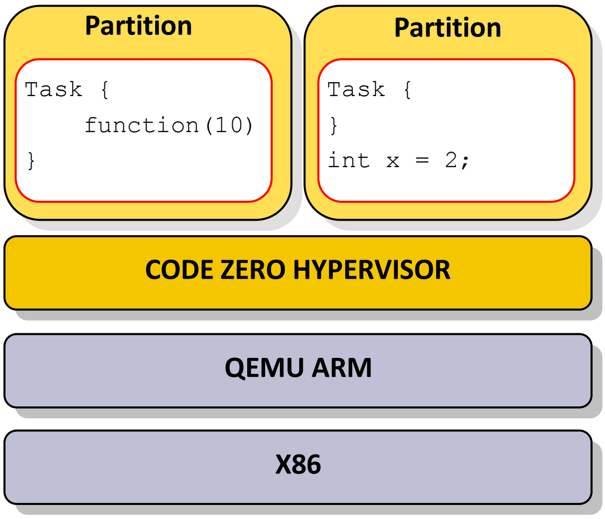


Fig.4. Fault Injection Architecture using Hypervisor

The platform used to launch was a PB926-Versatile. During our experiments, this board was not available, so we decided to emulate it with QEMU, as shown in Figure 4.

5.2 Global Variables Fault Injection

CodeZero provides the notion of partitions for virtualization. Each partition implements an isolated execution environment with its own set of resources (threads, address spaces, memory resources…). Access to virtualized resources is done through hyper calls from partition to hypervisor. In a first part, we choose to trigger fault injection on a partition when the partition itself makes a hyper call as described in Figure 5.

The steps used for this fault injection are:

- Compilation of targeted applications
- Reading the virtual address of the targeted global
- Starting the hypervisor
- On application hyper call: fault injections are done from the hypervisor on the application targeted global variable

In this solution, we notice that the injector is integrated within the hypervisor. Such a solution induces some modifications in hypervisor code, which is not always possible. Consequently, we tried to implement the injection monitor within a second partition (cf. Figure 6). This specific partition is in charge of injecting faults on other partition using hyper calls. The algorithm used is summarized in the following steps:

- Compilation of targeted applications
- Reading the virtual address of the targeted global
- Starting the hypervisor
- The injection partition makes an “injection fault hyper call“ with virtual addresses of targeted global variables as argument
- The hypervisor handles the hyper call, maps the address space of the targeted partition, updates the global variable value and re-maps the address space of the “fault injection partition”.

Notice that this solution adds an overhead since the address space swaps are costly.

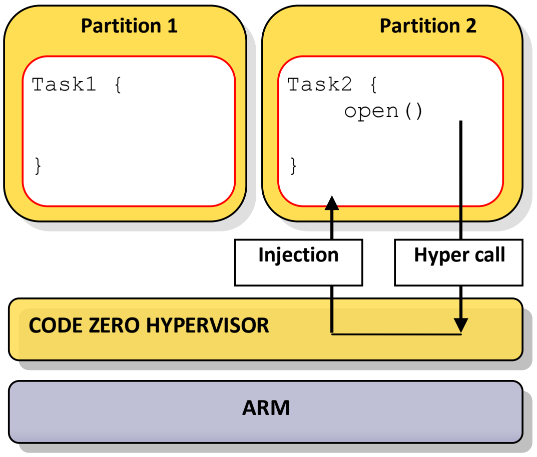


Fig. 5. Inter Partition Fault Injection with Hypervisor

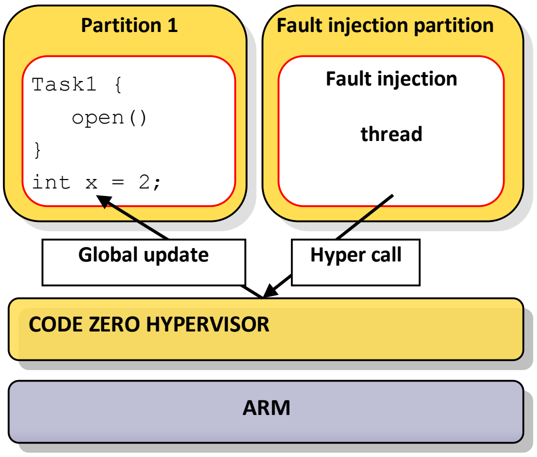


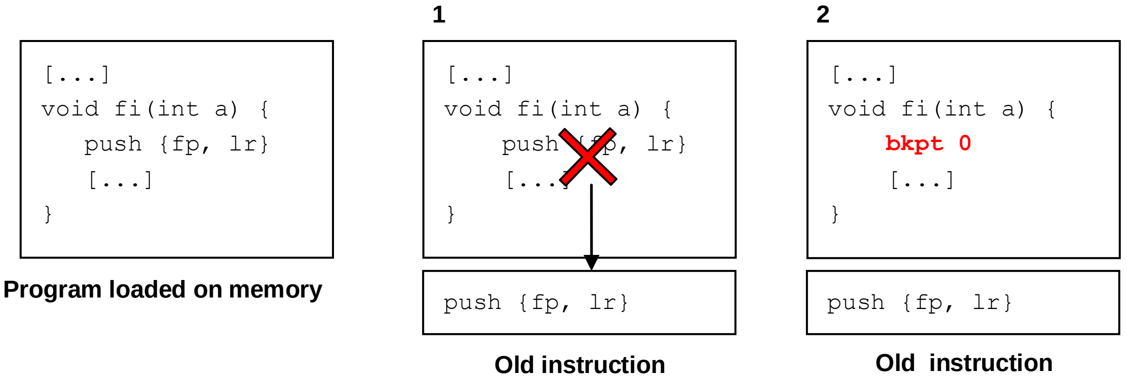
Fig. 6. Intra fault injection using hypervisor

5.3 Function Argument Fault Injection

This fault injection technique is similar to that used in section 4.2. Hardware breakpoints on targeted functions are used to inject faults. When a breakpoint is triggered, the fault is injected in the stack and in specific registers before normal execution resumes.

The main difference between “Function argument injection” technics on “QEMU” and “Function argument injection” technic on “CodeZero” is the following: with QEMU we can use the software debugging layer implemented on the emulator while in “CodeZero” we had to use the hardware debugging layer.

We propose to detail, by an illustrated example, the operation of hardware debugging layer on arm platform. The first argument of a function “fi”, executed on an ARM platform, is targeted by a fault injection. First we save the first instruction of “fi” function (Figure 6, Step 1) and replace it by a breakpoint instruction (Figure 6, Step 2).



```
[...]
void fi(int a) {
    push {fp, lr}
    [...]
}
```

```
[...]
void fi(int a) {
    push {fp, lr}
    [...]
}
```

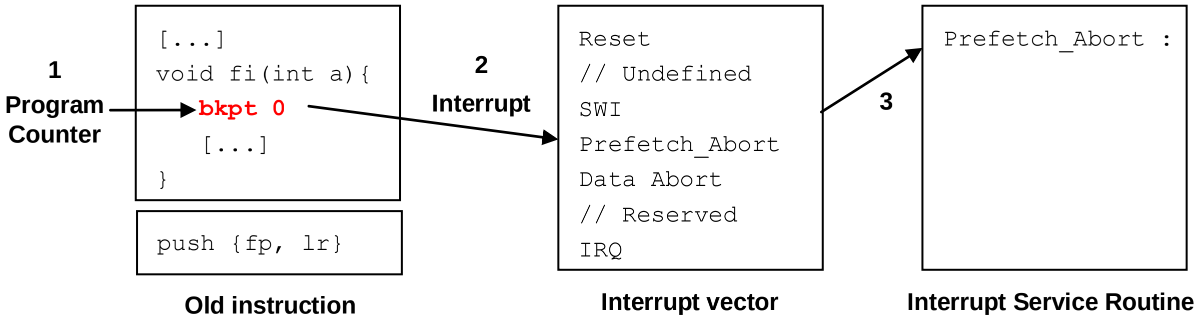
```
push {fp, lr}
```

```
[...]
void fi(int a) {
    bkpt 0
    [...]
}
```

```
push {fp, lr}
```

Fig. 6. Placing a breakpoint

When the function is called, the processor executes the breakpoint instruction (Step 1, Figure 7) and raise a *Prefetch Abort* interrupt (Step 2, Figure 7) handled by the *Prefetch Abort ISR* (*Interrupt Service Routine*, see Step 3, Figure 7).



```
[...]
void fi(int a){
    bkpt 0
    [...]
}
```

```
push {fp, lr}
```

```
Reset
// Undefined
SWI
Prefetch_Abort
Data Abort
// Reserved
IRQ
```

```
Prefetch_Abort :
```

Fig. 7. Interrupt

The *Prefetch Abort* ISR perform the fault injection by updating the processors register that contains the first argument of the “fi” function (Step 1, Figure 8) and replace the breakpoint instruction by the old instruction(Step 2, Figure 8).

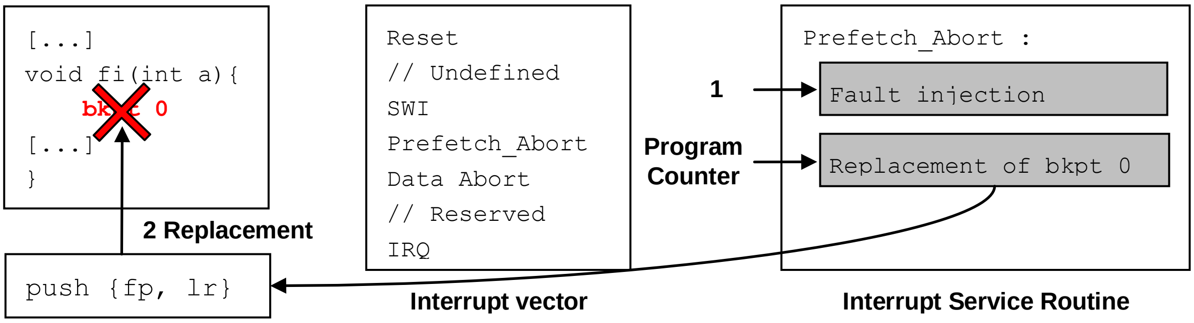


Fig. 8. Fault Injection

The fault injection handler returns the control to the “fi” function that continues its execution (Fig. 9).

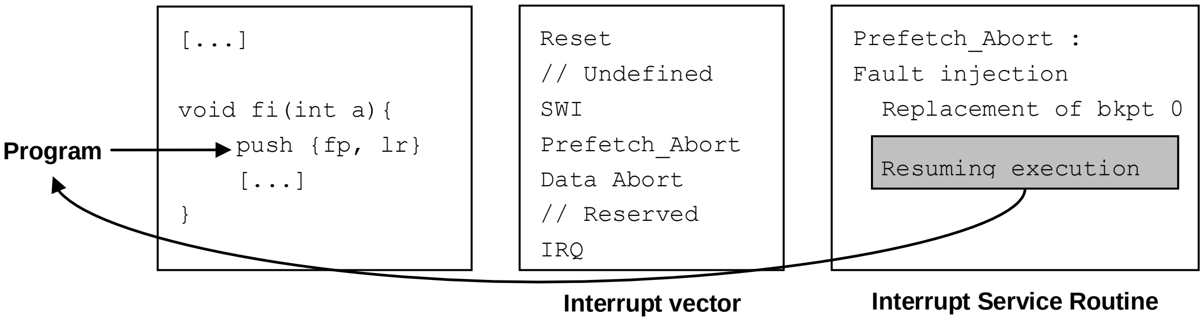


Fig. 9. Resuming Execution

The difference between software debugging layer implemented by *QEMU* and hardware debugging layer implemented is linked to timing issues. On one hand, *CodeZero* adds a significant CPU time overhead, with no control on the guest OS clock. This can lead to uncontrolled time drift. On the other hand *QEMU* freezes the guest virtual machine until the injection is done. The virtual machine does not realize that it is frozen.

6 Conclusion

We briefly presented fault injection techniques and described how virtualization can be used to inject mutants in memory without altering the source code of the applications.

Let us remind that fault injection cannot be done without a previous knowledge of the application behavior, which is an important hypothesis of this technique. However, fault injection supposes that we entirely know the software functional and dysfunctional specifications. This is necessary to observe any abnormal behavior when injecting faults.

We also focused on two main application fault injections types: the global variables and the functions parameters. During our experiments, we noticed that this injection technique has some limitations. First, it is sensitive to optimizations that are applied to the code. For instance, when options like "*inlining*" are enabled, function calls are included in the code and fault injection on their arguments becomes impossible. In addition, this technique is non deterministic. In fact, the state of the program when the breakpoint is set affects the result of the injection. For example, if a function is called twice in a code we cannot determine on which call the injection will be performed. We identified several solutions to figure out this issue. It generally consists in synchronizing fault injection with the running application. The purpose is to trigger the injection according to the inputs and the internal state of the program.

The described techniques have been tested on a proof on concept applications with safety requirements (ASIL C for instance). It could be possible to extend our work to additional fault injection types and to implement a framework to assess real-time properties of the applications especially the overhead introduced by fault injection techniques.

Bibliography

[1] Jean-Claude LAPRIE. *Guide de la Sûreté de fonctionnement*. France : Cépaduès, 1996

[2] ISO/FDIS 26262. *« Véhicules routiers - Sécurité fonctionnelle des systèmes électriques et électroniques »* Final Draft International Standard ISO/FDIS 26262 : 2010

[3] IEC 61508. *« Functional safety of electrical/electronic/ programmable electronic safety-related systems »* IEC 61508 Parts 1-7, First edition 12/1998.

[4] Karim HADJIAT. *Evaluation prédictive de la sûreté de fonctionnement d'un circuit intégré numérique*. Thèse Micro et Nano Electronique. Grenoble : Institut national polytechnique de Grenoble, 2005, 142 p.

[5] Mei-Chen HSUEH, Timothy K. TSAI, Ravishankar K. IYER. « Fault Injection Techniques and Tools », *IEEE Computer*, 1997, vol.30, n°4, p.75-82.

[6] AUTomotive Open System Architecture. *AUTOSAR* [en ligne]. http://www.autosar.org/.

[7] GENIVI Alliance. *GENIVI Alliance*. Available on : http://www.genivi.org/.

[8] ARINC. *« ARINC 653 Avionics Application Software Interface »,* ARINC : 2010

[9] IRCCyN. *Trampoline - OpenSource RTOS project* Available on : http://trampoline.rts-software.org/ .

[10] ARMadeus systems. *APF27* [en ligne]. Available on : http://www.armadeus.com/english/products-processor_boards-apf27.html .

[11] B-Labs. *Codezero*. Available on http://l4dev.org/ .

[12] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, Andrew Warfield. « Remus: High Availability via Asynchronous Virtual Machine Replication », *USENIX symposium*, 2008

[13] Yoshiaki Tamura, Koji Sato, Seiji Kihara, Satoshi Moriai. « Virtual Machine Synchronization for Fault Tolerance », 2008

[14] Yvan ROCH. « QEMU : Visite au cœur de l'émulateur », *Linux Magazine*, Mars 2012, n°147, p.6-27.

[15] Yvan ROCH. « QEMU : Comment émuler une nouvelle machine? Cas de l'apf27 », *Linux Magazine*, Avril 2012, n°148, p.48-70.

[16] *QEMU– Open Source processor emulator*. Available on :http://wiki.qemu.org/