STUDIA INFORMATICA 2012
Volume 33 Number 4 (109)

Sławomir CHYŁEK, Marcin GOLISZEWSKI
Warsaw University of Technology, Institute of Computer Science

QEMU-BASED FAULT INJECTION FRAMEWORK

Summary. This article presents the QEFI: a QEMU-based fault injection framework. It presents basic design principles behind the created framework, its implementation details and some results of experiments conducted to prove its utility. The novelty of the implemented software is outlined and compared with solutions presented in the literature.

Keywords: QEMU, fault injection, emulation, embedded systems, Linux, ARM

SYSTEM WSTRZYKIWANIA BŁĘDÓW OPARTY NA QEMU

Streszczenie. Artykuł ten prezentuje QEFI: system wstrzykiwania błędów oparty na QEMU. Przedstawia on podstawowe założenia projektowe stworzonego systemu, jego szczegóły implementacyjne oraz wyniki niektórych eksperymentów dowodzących jego użyteczności. Nowatorskie aspekty stworzonego oprogramowania zostały wyszczególnione i porównane z pracami znanymi z literatury.

Słowa kluczowe: QEMU, wstrzykiwanie błędów, emulacja, systemy wbudowane, Linux, ARM

1. Introduction

Dependable computer systems are becoming more and more important branch of modern information technology industry. The use of such systems range from data centers where every service unavailability means money-measured loss to embedded systems, such as anti-lock braking systems, where every failure may have cost in human live. Modern microprocessors, due to increased density of transistors, are getting more and more susceptible to external conditions like temperature or electromagnetic field – thus the number of faults (like bit flips, stuck at 0s or stuck at 1s) is increasing [8]. To increase the

dependability, many techniques are proposed and evaluated in following areas: testing, error detection, error prevention, recovery and fault injection. This paper will present a novel approach to model-based fault injection: a software fault injection (SWIFI) tool that is capable of injecting faults into an unmodified operating system while most of tools focus only on user-space applications.

The presented solution is called QEMU Fault Injector (QEFI) and it is based on the QEMU computer system emulator. QEFI design goal was to provide a tool to automate fault injection campaigns. Utilizing a model instead of a real device introduces major improvements to the testing process: no real device is needed, the system under test (SUT) is easily manageable and the solution as a whole is scalable. However, these improvements come at a price of performance overhead and more complex environment setup, since evaluating software cooperating with hardware devices requires implementation of these devices' simulators.

The research in question has been focused on the ARM architecture. This decision was motivated by growing popularity of mobile devices and embedded systems that are more and more often assigned to mission critical activities like Firmware Over The Air (FOTA) or monitoring user's health parameters. These devices are usually exposed to unfavorable conditions like sudden change of temperature or high humidity which increases the risk of errors. The market of embedded systems is also very diverse since hardware units are entering the market every few months. The ability to evaluate software in simulated environment is an advantage that allows to operate even if the hardware is still in the production phase or the number of real units is limited.

The rest of the paper is organized as follows: overview of the presented solution, related work, detailed QEFI architecture and presentation of experiments' results.

2. QEFI Overview

QEFI was designed as a tool for system-wide and kernel-based faults injection. The chosen approach allows for creating a fault injection framework focused on simulating hardware faults and testing software reactions to them. The project's aim was to create a tool for developers and testers to possibly replace their hardware target devices, in at least a part of the software development process, with emulated environments. This would allow them to benefit from the controllability and observability of the proposed QEFI framework resulting in the possibility of testing software interactions with hardware operations hard to achieve while using a real target device. Most notably, the aim of QEFI framework is to allow for simulating different kinds of hardware failures and environmental issues, such as (among

others) limited connectivity, physical hardware modules failure, internal device connection errors, CPU errors etc.

Utilizing QEMU is a way to evaluate reliability of kernel space code. This may be achieved by injecting faults into processor's registers, data in memory or the executed code. Usually, the test scenarios are targeted only at user space applications as most injection frameworks are implemented in kernel space (a disturbance of kernel code would also disturb the observer) or require a standalone observer [1, 4]. Allowing for kernel-level injections has a major research value because in case of QEMU we are able to test the kernel's robustness using an external injection framework. This utilization of QEFI is complementary to injectors that disturb user space processes since QEMU is aware only of working kernel image and doesn't have any ability to trace user space applications in generic manner.

QEFI works are focused on creating an extensible and versatile framework for running fault injection test campaigns. Replacing a real device with a model introduces another benefit of easy replication of environment under test. This feature may be utilized to scale testing campaign and significantly improve testing coverage.

The rest of this section outlines strong points and weaknesses of the implemented solution. The solution evaluation is divided into Model fault injection, QEMU and QEFI subjects to present a full picture at different abstraction levels.

2.1. Model fault injection

Utilizing models in the fault injection process is beneficial due to easy model replication and extended controllability. Models can be implemented as microarchitecture simulation, instruction level simulation or even virtualization. In all of these techniques preparing copies of a testing environment is cheap compared to manufacturing a real-device environment, thus it is possible to perform testing on a large scale in an automatic manner. This is a major advantage since testing campaign execution time may be reduced and its coverage may be extended.

Testing with model-based fault injection techniques can be conducted at any abstraction level – it just depends on models' level of detail – e.g. accelerometer application showing current acceleration to the user can be run on a model of OS, where syscalls are mocked to return current acceleration; alternatively, a model of hardware can be utilized in order to test the OS kernel module as well. This extends the developer's or tester's set of tools and simplifies the testing procedure – without such tool the tester, to test how application displays large acceleration values, would have to be really exposed to such acceleration, which could be inconvenient.

Model approach has two major drawbacks: model environment preparation can be expensive and the test results may be inaccurate. These issues are related to the need of

preparing a model that mimics the target platform as accurately as possible. To reduce the cost of model environment setup, it is recommended to use some products already present on the market, like simulators or virtualization software, and adjust them to emulate the target platform. To address the second issue, it is recommended to run a series of tests in tandem on a real device and in model environment to ensure that model's accuracy is satisfactory. However, it may be safely assumed that as a device becomes more popular on the market its simulators are more accurate.

2.2. QEMU

QEMU is a simulation platform which was chosen as a base of the presented model-based fault injection concept implementation. This decision was driven mostly by two factors: reasonable performance and availability under an open source license. With QEMU platform it was possible to implement the injection of various kinds of faults, e.g.: CPU registers modification, data in RAM modification, storage devices' faults or network interfaces faults. The architecture of QEMU allows to easily extend the range of possible faults by modifying its Tiny Code Generator (TCG) which is responsible for executing emulated system code or introducing modifications in virtual devices' state and behavior.

QEMU is equipped with a shell user interface, which has been utilized as a point of control. This interface allowed both to implement user manual control over fault injection experiment and an automatized framework used to conduct experiments grouped in campaigns. Another QEMU's advantage is out-of-the-box interaction with major developer's tools – e.g. GDB integration, which is used by QEFI to provide an alternative user interface for fault injection configuration and triggering.

2.3. QEFI

QEFI framework has been designed as an environment for configuring and running fault injection experiments grouped in campaigns. An experiment is a single run of an emulated SUT with an injected fault. It can be described as a sequence of the following phases: booting the emulated device, setup of preconditions, injecting a fault, interaction with the system to see how/if the fault propagates and collection of post-experiment data. A campaign is a series of experiments conducted in an automatic fashion. Support for the campaign mode is crucial since conducting several experiments in a row manually by the operator would be very time consuming.

To accomplish the aforementioned requirements, QEFI has been implemented as a Python module with QEMU as a backend. This decision has several advantages:

- clear components' responsibility separation – components can be easily isolated and tested independently,
- script-like definition of experiments,
- great flexibility of interaction with the system – user can work with the system in an interactive mode, record issued commands and replay them in the script parameterized according to his/her needs,
- configuration can be generated dynamically – it is even possible to setup next experiments based on previous experiments' results (e.g. to stress test scenarios that were judged to be interesting according to some rules),
- architecture is prepared for multiprocess execution – all experiments are conducted in separate working directories thus they follow data separation pattern and are ready to be run safely in parallel on machines that are capable of greater load.

3. Related work

Fault injection as a tool for evaluating software robustness is widely described in the literature. A vast variety of solutions have been implemented over the years [2]. Software fault injection techniques can be divided into categories according to different characteristics. SWIFI frameworks can be categorized by execution environment – a real device or a model. The former category includes injectors utilizing debugging/trap API to alter program's execution [6, 10], modified kernel [1, 4, 11] or even modified BIOS program [16]. Injection tools in simulated environments, as stated in previous section, operate at different abstraction levels: from low-level hardware models [7, 15] to virtual instances of the operating system [3].

QEFI represents the simulation-based approach. As opposed to [3], it is based on an instruction-level simulator thus has the advantage of emulating different architectures and doesn't require any modifications of the software under test. The quality of this approach is legitimized by the works presented in [12] where the very same real device fault injection framework was run on a real device and an emulated one with similar results.

The importance of scalability of fault injection frameworks has been presented in [13] and [14]. These works influenced the design of QEFI – the adaptability to run in distributed environments was a major goal since it is a key to reasonable performance in evaluating complex software.

4. QEFI architecture

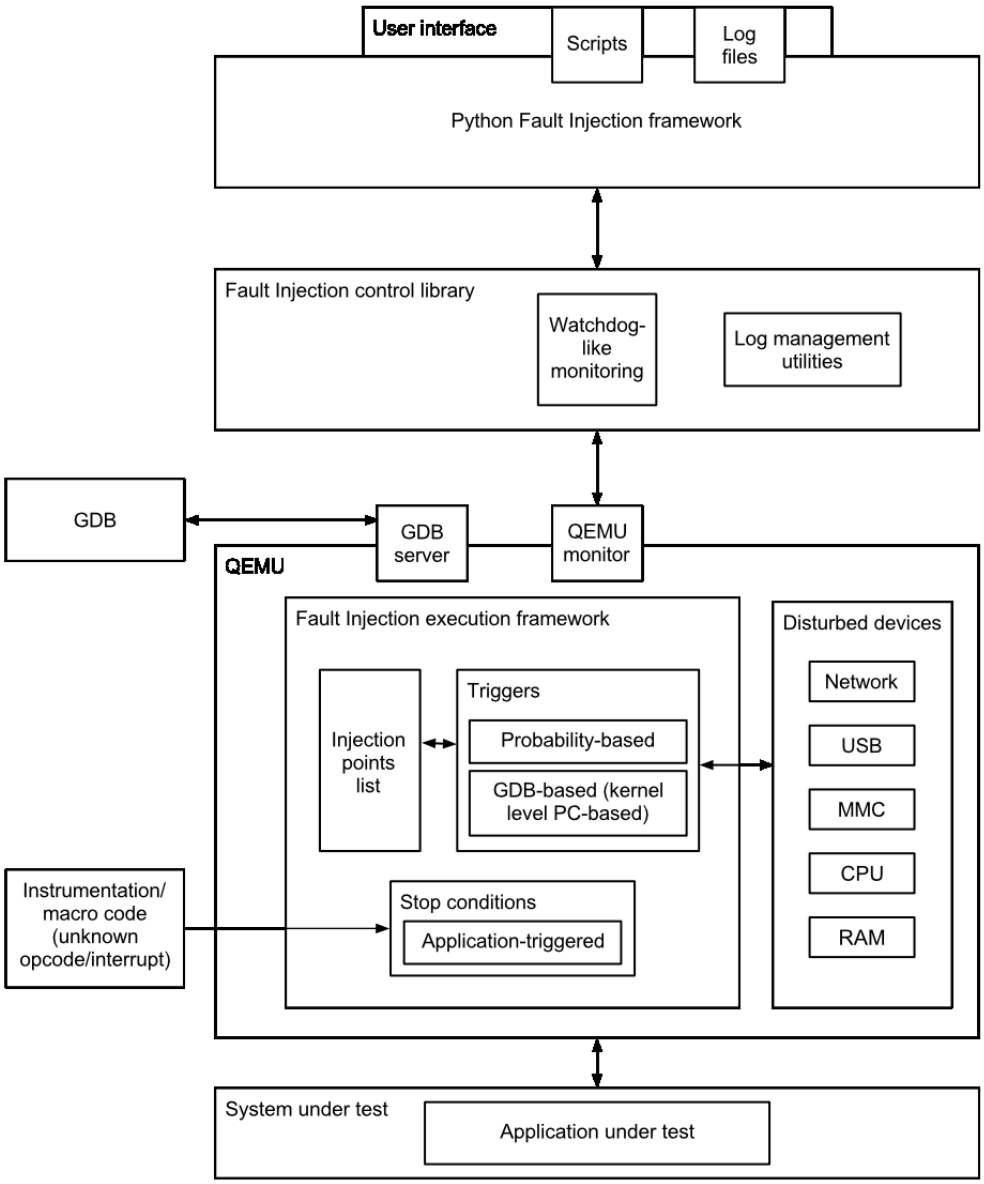

Fig. 1. QEFI framework architecture diagram
Rys. 1. Diagram architektury QEFI

The architecture of the implemented QEFI system is presented in fig. 1. The most important architectural concepts of this design are its extensibility and versatility. That is why the architecture is a multi-layer, multi-module structure. Its three major parts are: the fault injection control framework, the QEMU emulator itself and a fault injection control library used as a proxy between the two other layers, responsible mainly for communication and QEMU execution management. The reason of separating the controlling layer from QEMU is the urge to modify QEMU as little as possible. Limited number of modifications inside QEMU code allows for higher compatibility with its future versions and easier management

of the created patches. QEMU main modifications are: the fault injection code in the emulated devices, management of so-called injection points. Injection points describe how a fault should be injected into SUT. Injection points consist of:

- disturbed device (e.g. CPU, RAM, USB),
- type of injection (e.g. bit flip in memory, lost token),
- triggering conditions (e.g. specific program counter «PC» value, random injection with specific probability, specific device activity or external application triggering).

The created framework allows for creating injection points of various types – described in details in a further section.

5. Implementation Details

5.1. QEFI Python interface

The Python-based script API is the main user interface of QEFI – the one recommended to be used for most fault injection experiments. It provides the user with the ability to define different kinds of experiments, control their run, gather results and add custom modifications to any of the steps involved in the process. The experiments' control is easy for the end user thanks to the object-oriented design of the QEFI interface: the user is given a Python object representing a QEMU instance and may freely interact with it. The user may control QEMU with monitor interface (e.g. to connect or disconnect USB mass storage device) and interact with the emulated system using an API similar to the one provided by a popular UNIX tool called expect: it allows to read/write from/to emulated system's serial output/input, create conditions on the output's availability etc.

5.2. Experiments and campaigns

An experiment is the most basic form of an operation of QEFI. It's defined as a simple series of steps to be executed which lead to the desired application run scenario, fault injection and post-fault logs collection. The following elements are to be expected to appear in most experiments:

- QEMU run control commands,
- application run control commands,
- fault injection points management commands,
- results gathering commands,
- results processing commands.

An experiment is written in a form of Python script that may be prepared during interactive session with QEFI. The script may be parameterized in terms of changing executed application's input data or moment/type of fault injection. Such script can be then run in a batch mode for different parameters' values. The series of experiment runs is called a campaign and is designed as a way to gather more general statistics about SUT's susceptibility to faults.

5.3. Logging

During the course of QEFI's implementation, many different logging techniques have been considered. As implementing a common logging mechanism for such a heterogeneous environment (consisting of separate pieces of software implemented using completely different programming techniques and with different design principles in mind) would be very hard and could result in a very inefficient solution, it has been decided to adopt the "every man for himself" strategy. The responsibility for implementing logging has been passed as far down as possible – to the lowest layers of the software, i.e. to the point where the need for logging arises. This way every module may define its own way of logging which best suits it's needs. The QEFI framework focuses on providing some logs management utilities. These are:

- providing a directory to gather the logs in – the current working directory is always bound to be a safe place in which the application may store it's logs,
- ability to timestamp the logs with synchronization marks to ease their further processing and browsing.

5.4. QEMU

Although not directly visible to the user, the QEMU part of QEFI implementation is its most important module as it implements the logic allowing to perform fault injection in its fault injection execution framework. It serves several roles crucial for the fault injection process operation:

- holds the lists of injection points to be executed and stop conditions to be triggered,
- provides utility functions and C language macros to ease the implementation of device-specific faults injection,
- detects and handles stop conditions,
- supports user-interface (used mostly to interface the upper layers of QEFI framework).

This module may be interfaced in two manners: either using the QEMU monitor or using GDB. The QEMU monitor interface is used for several different purposes:

- command/response communication with the upper layers used for injection points management and direct fault injections (e.g. to the CPU registers and/or RAM memory),
- asynchronous notifications of QEMU's internal events – e.g. the occurrence of a stop condition,
- watchdog-style monitoring of QEMU's operation (in a challenge-response manner).

The GDB, due to its nature, is used in a simpler manner – only for defining fault injection points related to the operating system kernel being run by QEMU.

During development of the presented fault injection framework it has been also considered to integrate a QEMU's snapshot feature to speedup campaigns execution by restoring an emulated device to a checkpoint state instead booting it from the scratch. However, snapshot feature is not fully supported on some of QEMU's architectures (including ARM). Nevertheless, snapshot feature may be a potential source of optimisation to QEFI.

5.5. Injection points

Injection points are one of the most important terms introduced during QEFI design and development. An injection point is an entity which mimics the design of widely known break points and watch points. Four types of injection points may be distinguished:
Probability-based triggers – type of injection points which are triggered with user-defined probability in random moments of the system's execution.

- PC-based triggers – type of inject points most closely resembling watch points and break points. These are triggered when the execution reaches a specified PC value which enables the user to setup injections in some specified line of code (or a range of lines), e.g. in the operating system kernel.
- Application-triggered injections – injection points defined during the tested application's runtime in a "injection start" – "injection stop" commands manner. The commands are issued directly to the QEMU layer of the system as an invalid processor instruction execution or unknown interrupt triggering. A proposed workflow using this kind of injection triggering is using GCC's functions instrumentation feature to enable/disable faults injection on a per-function basis. Note that such fine-grained fault triggering can only be achieved at the cost of modifying the tested software which may not always be acceptable.
- Device-triggered injections – fault injection is triggered by devices' activity – e.g. when an interaction with USB mass storage device is observed faults are going to be injected (this assumes support from virtual devices modules).

5.6. Stop conditions

Stop conditions are a natural evolution of injection points aiming at giving the user full control of what actions to perform when the defined code point has been reached. The novelty of this concept lies in the method used for triggering the stop conditions: their main source is the tested application itself. QEFI's supplementary tools allow for instrumenting the tested application using opcodes not valid on the ARM architecture. These are being handled by the modified QEMU by stopping the virtual machine's execution, passing details about the stop condition encountered to the upper layers and waiting for user's actions. Such a case may be used for adding, removing, enabling or disabling injection points but may also be used to perform some more complicated actions, e.g. based on the current state of the application or on the results of previous experiments.

5.7. Disturbed devices

The implemented QEFI prototype supports disturbing the operation of the following virtual devices:

- USB OHCI controller,
- USB mass storage device,
- Realtek 8139 Ethernet card,
- PL181 MMCI controller.

Different fault types are supported to be injected for different virtual devices – table 1 summarizes the implemented possibilities.

Table 1

Supported injected fault types

	Packet drop	Buffer/packet modification	Packets reordering	Packets duplication
USB OHCI	✔	✔	✖	✔
USB MSD	✔	✔	✖	✖
RTL8139	✔	✔	✔	✔
PL181	✔	✔	✖	✔

Apart from the above faults, the USB MSD supports also two additional types of faults: incoming token dropping and outgoing token dropping. These are specific errors valid only for USB devices and allowing to fine tune the fault injection scenarios tested.

The implementation of these injection points has been done by patching the virtual device's source code and adding fault injection instructions in appropriate places. Such a method, which may be considered not to be of clean design, is caused by the lack of any

hooks-like mechanism in QEMU. As a side effect, it allows for great controllability as the injection point's location is chosen with line-based granularity.

5.8. CPU/RAM injections

Apart from the above possibilities to inject errors into specific devices composing a computer system, it's also possible to disturb the operation of its core elements: the CPU and RAM.

CPU operation disturbance is based on its registers values modification. The user is presented with a possibility to freely overwrite any of the 16 ARM processor registers to influence the system's operation on the lowest level. The possible use cases include function arguments or return value altering, PC register modification to change the program flow, control over the operations being performed by changing computed values stored in registers and many others.

A similar feature is available to disturb the RAM. It's possible to freely modify any given memory area – specified either by its virtual or physical address to allow for process-based or kernel-based injections, respectively. This tool gives the user even more control over the program's execution – e.g. a possibility to dynamically change its variables' values, control the branches it executes, disturb the calculations it performs etc.

These two injection methods, although very low-level and requiring special care to perform interesting actions, may be considered to be an example of QEFI's extraordinary controllability capabilities. It's an example of how useful simulation-based fault injection may be – achieving similar features on real hardware device would be very difficult (if possible at all).

5.9. TCG modifications

The most crucial modification made to QEMU's source code is the modification of its Tiny Code Generator. Two features needed support from TCG-level patches: the stop conditions and PC-based injection points. Both of these are implemented as additional operations added to the code generated by TCG. Please note that these additions, although very useful in performing fault injection experiments, may break one of the core features of QEMU – its speed may be decreased.

5.10. GDB integration

GDB usage is one of the methods to define injection points. Its advantage over direct definition of the injection points in test scripts is the ability to define them on source code

line basis. This gives the developer the possibility to test a given part of Linux kernel's code by injecting faults only when needed.

To integrate QEMU fault injection execution framework and GDB, the support for GDB server protocol (already present in QEMU) has been used. The protocol has been enhanced to support not only break points and watch points but also injection points. The GDB has been modified to support additional commands to facilitate the fault injection possibilities when attached to a remote GDB server. The injection point created in QEMU's fault injection execution framework using GDB differs slightly from other injection points – it's being executed only when an appropriate PC register value (translated by GDB from the source code location provided by the user) is reached. Other parts of the injection points handling and the fault injection process is exactly the same.

6. QEFI experiments

During the course of QEFI's implementation several different experiments have been performed to confirm its implementation correctness and its utility for the purpose it has been designed for. Note that extensive usage of QEFI for testing purposes has not yet been done so the presented results may be considered as preliminary. The performed experiments demonstrated some flaws – both of the Linux kernel (which is considered as the application under test) and of the QEMU. These results are briefly described in the following sections.

6.1. Recovery scenarios

QEFI approach is an adequate technique to evaluate recovery mechanisms present at the kernel level of operating systems. Conducting experiments with virtual devices gives the ability to observe not only direct effects of faults but also any attempts to restore original condition of the system.

One of the experiments which has been conducted during QEFI development was targeted at injecting faults into USB OHCI (Open Host Controller Interface) emulated device. The faults were distortions in OHCI registers influencing its communication flow. It has been observed that in some scenarios the fault was detected and the Linux kernel attempted to reset the device. It was a basic recovery technique that was supposed only to restore communication via USB and was not aimed at restoring its state or the previous communication flow.

More sophisticated recover techniques were observed when injecting faults into USB MSD (Mass Storage Device), which in our case was a virtual pendrive formatted using ext3 filesystem. The faults were introduced into the payload of USB tokens, which means altering

ext3 filesystem internal data and stored files contents. The ext3 filesystem is a journaled file system, thus after detecting an error a recovery procedure is triggered. Four different results of recovery mechanism's application were observed:

- successful recovery procedure,
- filesystem was restored but files were altered,
- filesystem was restored but it was re-mounted read-only,
- operating system was unable to recover the filesystem (i.e. the pendrive rendered unusable).

These results prove that QEFI can be successfully utilized as a tool for evaluating recovery procedures implemented at the kernel-level. Ease of use and no need to alter the evaluated code can support pointing out critical aspects of neutralizing effects of failures.

6.2. Crash scenarios

During the conducted experiments, crash scenarios were also observed. It has been spotted that after modifying the registers of the USB OHCI virtual device, Linux kernel sometimes didn't validate the data read from these registers. This resulted in invoking "kernel oops" and "kernel panic" procedures, which are considered as irrecoverable errors – either for the given system module or for the system as a whole, respectively.

Another example is writing an arbitrary value into the register #16 of the emulated ARM processor which is the PC register. When the register's value was modified during user space process execution (injecting a fault in the user space process was possible thanks to the stop conditions), it lead to a segmentation fault if the value was out of the current process' memory area. It was an irrecoverable error for the process but the operating system remained intact. On the other hand, when the PC value was modified during the system boot (without distinguishing if current processor's context is user-space or kernel-space), it happened that the kernel was trying to execute some random piece of code which resulted in a "kernel panic".

Described scenarios prove that QEFI is capable of introducing irrecoverable crash scenarios. Ability of injecting such faults is valuable since it makes possible to prepare procedures for such conditions and test theirs effectiveness.

6.3. Application-oriented scenarios

QEFI may be used not only to disturb kernel logic but also user-space applications' logic. During the implementation of various injection mechanisms some simulated scenarios were conducted to test if these mechanisms are functional. This section describes such exemplary tests.

To test a scenario of a faulty random access memory (RAM) the program presented in Appendix 1 has been prepared. It is a sample program which was prepared on purpose but in a real testing campaign the modified memory area should be selected by an expert. This program presents the usage of the stop condition technique (embedded assembler code) to pause the execution at a given point. Thanks to this, we were able to overwrite the memory value using the QEMU monitor interface. The output of the program run in such a case is presented in Appendix 2. It shows that QEFI can be utilized to externally modify process' environment.

A similar experiment has been conducted with overwriting the PC value of a variant of the standard UNIX ls tool augmented with QEFI instrumentation. It was possible to set the value of the PC to point the beginning of the program thus executing it twice. The output of this experiment is presented in appendix 3. Errors visible at the end of the output are caused by abnormal stack state (overwriting the PC register disturbs the ordinary flow of pushing and popping values from the stack).

7. Future work

Creating the QEFI fault injection framework is the first step to evaluate operating systems' susceptibility to faults. The plans of QEFI's authors include conducting subsequent experiments where major focus will be put on working with real-world applications. Planned experiments range from stress-testing the Linux kernel in a search for yet unknown bugs which may influence its stability to end-to-end testing of software stacks which are commonly used in the real world – such as GUI applications, web servers, mail transfer agent servers etc.

QEFI development will also be continued in the future. The most notable focus areas are: implementation of the support for x86 architecture and adjusting the created software to be run in environments optimized for massively parallel execution. To achieve this QEFI will be not only optimized for multi-thread and multi-process execution but also the ability to distribute the campaign to many separate systems will be introduced. Also, some performance tests are planned to show if such approach may influence the testing process in a noticeable manner. All the future developments and all gathered results will be presented in separate articles.

Acknowledgements

These works were carried out in cooperation with the Samsung Poland Research & Development Center in Warsaw as a part of joint project “Adapting fault injection techniques to improve Samsung mobile products”. This project is co-financed by the Polish Agency for Enterprise Development as a part of Measure 1.4 Support for goal-oriented projects within the Priority Axis 1 and Measure 4.1 Support for the implementation of R&D results within the Priority Axis 4 of the Operational Programme Innovative Economy.

Appendices

Appendix 1: Faulty RAM test program

```
#include <stdlib.h>
int main(int argc, const char ** argv)  {
   int * val = malloc(sizeof(int));
   printf("Address is %p\n", val);
   *val = 99;
   printf("Value is %d\n", *val);
   printf("Pre involving swi\n");
   __asm__ (
       "swi $0xad\n\t"
   );

   printf("Post involving swi\n");
   printf("Value is %d\n", *val);

   free(val);

   return 0;
}
```

Appendix 2: Faulty RAM test result

```
debian-armel:~# ./mem
Address is 0x11008
Value is 99
Pre involving swi
Post involving swi
Value is 42
```

Appendix 3: PC value altering experiment result

```
debian-armel:~# ./badls.instr
.
..
badtee
badls
badls.instr
..
badtee
badls
badls.instr
Unknown cp14 read op1:7 crn:7 crm:4 op2:1
Unknown cp14 read op1:7 crn:7 crm:12 op2:1
```

```
Unknown cp14 read op1:7 crn:7 crm:4 op2:1
Unknown cp14 read op1:7 crn:7 crm:13 op2:7
Unknown cp14 read op1:7 crn:7 crm:9 op2:7
Illegal instruction
```

BIBLIOGRAPHY

1. Albinet A., Arlat J., Fabre J.: Characterization of the impact of faulty drivers on the robustness of the Linux kernel. International Conference on Dependable Systems and Networks, 2003, p. 867÷876.
2. Arlat J., Crouzet Y., Karlsson J., Folkesson P., Fuchs E., Leber G. H.: Comparison of physical and software-implemented fault injection techniques. IEEE Transactions on Computers, 2003, Vol. 52, No. 9, p. 1115÷1133.
3. Buchacker K., Sieh V.: Framework for testing the fault-tolerance of systems including OS and network aspects. Sixth IEEE International Symposium on High Assurance Systems Engineering, 2001, p. 95÷105.
4. Carreira J., Madeira H., Silva J. G.: Xception: a technique for the experimental evaluation of dependability in modern computers. IEEE Transactions on Software Engineering, Vol. 24, No. 2, p. 125÷136.
5. Chylek S.: Collecting program execution statistics with Qemu processor emulator. IMCSIT '09. International Multiconference on Computer Science and Information Technology, 2009, p. 555÷558.
6. Gawkowski P., Sosnowski J.: Developing Fault Injection Environment for Complex Experiments. IOLTS '08. 14th IEEE International On-Line Testing Symposium, 2008, p. 179÷181.
7. Gil D., Gracia J., Baraza J. C., Gil P. J.: Analysis of the influence of processor hidden registers on the accuracy of fault injection techniques", Ninth IEEE International High-Level Design Validation and Test Workshop, 2004, p. 173÷178.
8. Gracia J., Saiz L., Baraza J. C., Gil D., Gil P.: Analysis of the influence of intermittent faults in a microcontroller. DDECS 2008. 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, 2008, p. 1÷6.
9. Han A. H., Young-Si Hwang, Young-Ho An, So-Jin Lee, Ki-Seok Chung: Virtual ARM Platform for embedded system developers. ICALIP 2008 International Conference on Audio, Language and Image Processing, 2008, p. 586÷592
10. Kanawati G. A., Kanawati N. A., Abraham J. A.: FERRARI: a tool for the validation of system dependability properties. FTCS-22 Twenty-Second International Symposium on Fault-Tolerant Computing, 1992, p. 336÷344.

11. Madeira H., Costa D., Vieira M.: On the emulation of software faults by software fault injection. DSN 2000 Proceedings International Conference on Dependable Systems and Networks, 2000, p. 417÷426.
12. Murciano M., Violante M.: Validating the dependability of embedded systems through fault injection by means of loadable kernel modules. HLVDT 2007. IEEE International High Level Design Validation and Test Workshop, 2007, p. 179÷186.
13. Sosnowski J., Tymoczko A., Gawkowski, P.: An Approach to Distributed Fault Injection Experiments. PPAM'07 International conference on Parallel processing and applied mathematics, 2008, p. 361÷370.
14. Sosnowski J., Tymoczko A., Gawkowski P.: Developing distributed system for simulation experiments. Information Systems Architecture and Technology, 2008, p. 263÷274.
15. Svenningsson R., Eriksson H., Vinter J., Törngren M.: Model-Implemented Fault Injection for Hardware Fault Simulation. Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa), 2010, p. 31÷36.
16. Tröger P., Salfner F., Tschirpke S.: Software-Implemented Fault Injection at Firmware Level. Third International Conference on Dependability (DEPEND), 2010, p. 13÷16.
17. Weining Gu, Kalbarczyk Z., Iyer R. K.: Error sensitivity of the Linux kernel executing on PowerPC G4 and Pentium 4 processors. International Conference on Dependable Systems and Networks, 2004, p. 887÷896.

Wpłynęło do Redakcji: 21 lutego 2012 r.

Omówienie

QEFI jest implementacją jednej z technik analizy niezawodności za pomocą wstrzykiwania błędów – wstrzykiwania opartego na modelach. Celem omawianego projektu było stworzenie systemu wspomagającego analizę niezawodności systemów wbudowanych i mobilnych wyżej wymienioną metodą, tzn. poprzez wykonywanie eksperymentów na modelach tychże urządzeń.

Opracowane rozwiązanie zostało oparte na emulatorze QEMU. Emulator ten został wybrany ze względu na jego szybkość działania, a także rozszerzalność i możliwości adaptacji do specyficznych zastosowań. Architektura zaimplementowanego systemu jest wielowarstwowa i wielomodułowa – przedstawiono ją na rys. 1. Oprócz QEMU składają się na nią: system sterowania wstrzykiwaniem błędów opracowany jako zestaw skryptów i bibliotek języka Python oraz biblioteka sterująca wykonywaniem eksperymentów, której zadaniem jest zarządzanie wykonaniem procesu emulatora i zapewnienie interfejsu między

pozostałymi warstwami. QEFI pozwala na wstrzykiwanie błędów wielu różnych rodzajów – symulujących uszkodzenia lub pojedyncze błędy niektórych urządzeń zewnętrznych systemu (patrz tabela 1), rejestrów procesora czy komórek pamięci RAM. Większość z nich może być wyzwalana na wiele sposobów: probabilistycznie, na podstawie zawartości rejestru licznika instrukcji czy też na podstawie specjalnych instrukcji zawartych w testowanej aplikacji.

Artykuł przedstawia również wyniki eksperymentów wykonanych przy użyciu zaimplementowanego systemu. Oprócz zaprezentowania poprawności zastosowanej metody, a także użyteczności QEFI jako platformy wstrzykiwania będów, pokazują one przykładowe błędy oraz niedoskonałości odkryte w implementacji jądra systemu Linux. Zostały omówione zarówno wyniki testów, w których po wstrzyknięciu błędu została odzyskana pełna sprawność testowanego systemu, jak i takie, w których testowany system nie był w stanie dalej pracować poprawnie. Przedstawione eksperymenty mają na celu przedstawienie w pełni funkcjonalności QEFI, natomiast analiza niezawodności określonego oprogramowania wykracza poza zakres niniejszego opracowania.

Artykuł przedstawia również powiązane prace badawcze opisane w literaturze i w ich kontekście omawia właściwości systemu QEFI, a także przedstawia pobieżnie dalsze perspektywy rozwoju tego projektu.

Addresses

Sławomir CHYŁEK: Politechnika Warszawska, Instytut Informatyki, ul. Nowowiejska 15/19, 00-665 Warszawa, Polska, S.Chylek@ii.pw.edu.pl.
Marcin GOLISZEWSKI: Politechnika Warszawska, Instytut Informatyki, ul. Nowowiejska 15/19, 00-665 Warszawa, Polska, M.Goliszewski@ii.pw.edu.pl.