Experimental Analysis of the Errors Induced into Linux by Three Fault Injection Techniques

Tahar Jarboui, Jean Arlat, Yves Crouzet, Karama Kanoun

Experimental Analysis of the Errors Induced into Linux by Three Fault Injection Techniques

Tahar Jarboui, Jean Arlat, Yves Crouzet and Karama Kanoun
LAAS-CNRS, 7 avenue du Colonel Roche 31077 Toulouse Cedex 4 — France
{jarboui, arlat, crouzet, kanoun}@laas.fr

Abstract

The main goal of the experimental study reported in this paper is to investigate to what extent distinct fault injection techniques lead to similar consequences (errors and failures). The target system we are using to carry out our investigation is the Linux kernel as it provides a representative operating system. It is featuring full controllability and observability thanks to its open source status. Three types of software-implemented fault injection techniques are considered, namely: i) provision of invalid values to the parameters of the kernel calls, ii) corruption of the parameters of the kernel calls, and iii) corruption of the input parameters of the internal functions of the kernel. The workload being used for the experiments is tailored to activate selectively each functional component. The observations encompass typical kernel failure modes (e.g., exceptions and kernel hangs) as well as a detailed analysis of the reported error codes.

1. Introduction

Many supporting tools and techniques exist today that facilitate and automate the conduct of fault injection experiments [1]. In particular, due to its wide range of applicability and ease of implementation, the software-implemented fault injection technique (SWIFI) is now very popular. Nevertheless, although several studies have shown the wide range of faults that SWIFI can simulate (e.g., see [2-4]), more work is needed to better understand the actual behaviors that are induced by this technique. Indeed, such an understanding and characterization of the erroneous behaviors is mandatory so that fault injection experiments can turn into well-established dependability benchmarks. Previous work on robustness testing techniques where the target operating system is subjected to erroneous and/or stressful kernel calls offers a promising contribution for characterizing operating system behavior in the presence of faults [5-7]. Nevertheless, in order to facilitate the acceptance and the portability of such benchmarks, we have also to rely as much as possible on well-identified Application Programming Interfaces (APIs) to precisely specify how to perturb the operating system and enhance our understanding of the erroneous behaviors caused by the various injection possibilities that can be supported by SWIFI.

The ultimate aim of this work is the identification of techniques for generating faultloads that are representative of software faults that may impact operating systems. Indeed, operating systems (OS) are critical components of any computer system. Their malfunctions have a strong impact on the dependability of the global system. As a pragramatic approach we investigate to what extent distinct fault injection techniques lead to similar consequences. Indeed, in such case, it is worthwhile to select the technique that is easier to apply.

Even though perturbations affecting the operating system depend on the application domain (for example, space- systems are more exposed to hardware errors caused by radiations), all software systems are under the threat of software faults. This paper focuses on software faults.

The Linux kernel was selected as a target for our experiments as it provides a representative OS. It is featuring high level of controllability and observability thanks to its open source status. Fault model equivalence is established through the observations made after each experiment. We rely on the error detection mechanisms of Linux to draw our conclusions. We focus our analysis on the scheduling component.

The results presented in this paper reveal i) differences in the system behavior in the presence of either internal faults or faults applied at the API level and ii) some similarities for the two injection techniques used at the API level, from the point of view of the errors provoked, with a slight advantage for the bit-flip technique since it was able to provoke more distinct erroneous behaviors. Nevertheless, the set of invalid parameters used in these experiments could be enriched to include additional invalid cases. The first results corroborate the insights obtained in [5] for the Chorus microkernel, for which the failure modes induced by injecting bit-flips in the memory containing the code and data of the microkernel and at the API were different.

The rest of the paper is organized as follows. Section 2 presents the target system model. The experimental framework is presented in Section 3. Result analyses and comparison between the three injection techniques are presented in Section 4. Concluding remarks are drawn in Section 5.

2. Target system model

The target of our study is the Linux kernel. Four main entry points can be identified, through which Linux kernel functions are executed. Indeed, a switch to kernel mode can be triggered by: i) an interrupt issued to the CPU by a hardware device to indicate that it requires attention, ii) an exception signaled by a CPU because of an error, iii) a kernel call (or system call) issued by an application or iv) a kernel thread. The activation of kernel internal functions depends on these entry points but also on the current state of the kernel. In this paper, we concentrate on the third entry point: kernel calls issued via the API.

Based on the work presented in [8], and to facilitate the analysis of the Linux kernel, we decomposed it into five functional components: scheduling, memory management, synchronization, file system(s) management and communication. Each functional component is composed of elementary functions.

It is worthwhile to distinguish the elementary functions that are reachable from the API (kernel calls) from those that are not (internal functions). By modifying the *gcc* compiler, we were able to generate at kernel compilation a call graph for each kernel call. A call graph is composed of the elementary functions called by the considered kernel call. For each kernel call, we define depth levels. As an example, Figure 1 describes the call graph for the kernel call `sched_setscheduler` that has three depth levels. The "system_call" node is present in all call graphs associated with any kernel call. It represents the kernel call entry point.

The goal is to analyze the degree of similarities of the erroneous behaviors reported for the kernel as a consequence of fault injection at the first level (API) and in lower levels. The dots at the end of arrows represent the fault injection locations. The injection at the first level corresponds to ***external faults***, whereas fault injection in the lower levels (i.e., inside the kernel) maps to ***internal faults***. The injection techniques used are detailed in the next section. We detail in section 4 the results related to some scheduling kernel calls.

3. Experimental framework

In this study, three fault models are considered. A fault model is defined with respect to the fault type and to the fault location. The fault types used are ***bit-flips*** and ***invalid parameters***. We consider two locations, either the parameters of the targeted kernel call (i.e., external faults) or the parameters of the underlying kernel functions (i.e., internal faults). External faults mimic faults from the application level and they test the robustness of the kernel. Internal faults emulate various classes of faults such as those classified in the Orthogonal Defect Classification [9] (e.g., assignment, checking, interface, etc). In this paper, we consider only the interface class as indicated in Figure 1.

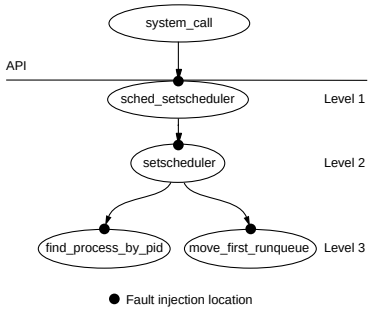


Figure 1- Call graph for sched_setscheduler kernel call

We associate a fault injection technique model to each fault model. The three considered injection techniques are thus: i) provision of API invalid parameters, ii) bit-flip in API parameters, and iii) bit-flip in internal function parameters.

The goal is to inject various faults and to observe and compare the resulting error sets. We have developed a versatile tool supporting the application of the three injection techniques. Each technique requires four main steps:

1) The kernel calls issued by the processes that the tool is tracing are intercepted. The tool uses Linux `ptrace()` interface and intercepts kernel calls in user mode as in [10] and [11]. The kernel call that is targeted by the fault injection experiment is thus interrupted.
2) A fault is injected according to the associated model (i.e., technique). The injection process ensures the synchronization between the fault and the workload and thus allows for result comparison for the three techniques.
3) The execution of the interrupted kernel call is resumed.
4) The system behavior is observed.

Table 1- Data type classes used by the scheduling component

Permission flag	*-1*	*0*	*ULONG_MAX (all bits to 1)*			
Integer	*INT_MIN*	*0*	*INT_MAX*			
Process identifier	*-1*	*0*	*INT_MAX*			
Read pointer	*Empty*	*-1*	*NULL*	*Non NULL*	*Freed*	*Random*
Write pointer	*Small (1 Byte)*	*Far (p + 4 MB)*	*NULL*	*Low (0x00000010)*	*Negative*	*Random*
Time pointer	*Negative*	*NULL*	*Random*	*INT_MAX*		

Figure 2 illustrates the experimental framework, which is based on a target and a host separate machines. The target kernel is installed on the *target machine* along with the injection tool. The hardware platform of the target machine is based on a Pentium III processor.

Both transient and permanent faults can be injected. A transient fault is automatically removed after its first activation, while it is only removed at the end of the experiment, if it is permanent. Two specific modules (invalid parameters database and internal sabotage controller) provide the capabilities that are specific to each kind of injection technique described in the next sections. The aim of the *host machine* (that is connected to the target machine through an Ethernet link) is to monitor the target machine and to reboot it with the adequate options in case of an application hang.

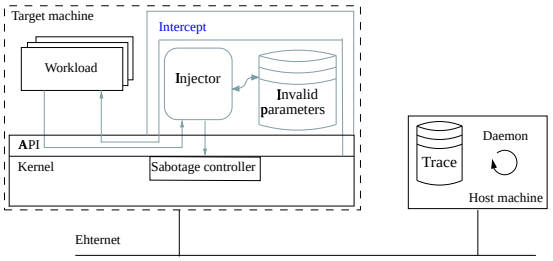


Figure 2- Experimental framework

3.1. Injection of external faults

We compare error sets caused by bit-flips (as for MAFALDA [12]) into system call parameters with those caused by invalid parameters (as for Ballista [6]). Only one parameter is corrupted per experiment, by either a single bit-flip or an invalid value. The injector module, shown in Figure 2, injects faults into the parameter values by either issuing exhaustive bit-flips (32 per parameter) or replacing them with invalid values. A set of invalid values for each data-type used in the Linux API is specified in a separate file (Invalid Parameters in Figure 2). Based on the work related to Ballista, and especially its online demonstration site, eight classes of invalid parameters are defined. Only six of these classes, presented in Table 1, are used to target the scheduling component. These values are either invalid or close to the limit of the valid domain; indeed, the goal is to stress the system as much as possible.

3.2. Injection of internal faults

This set of experiments targets the kernel internal functions that are not reachable from the API. The selected fault model consists in randomly injecting bit-flips in the input parameters of the considered function. According to the **Orthogonal Defect Classification**, these faults correspond to interface faults.

We distinguish the fault insertion phase from the fault-enabling phase. The fault insertion phase instruments the kernel code and is semi-automated. Code instrumentation is achieved in two steps:

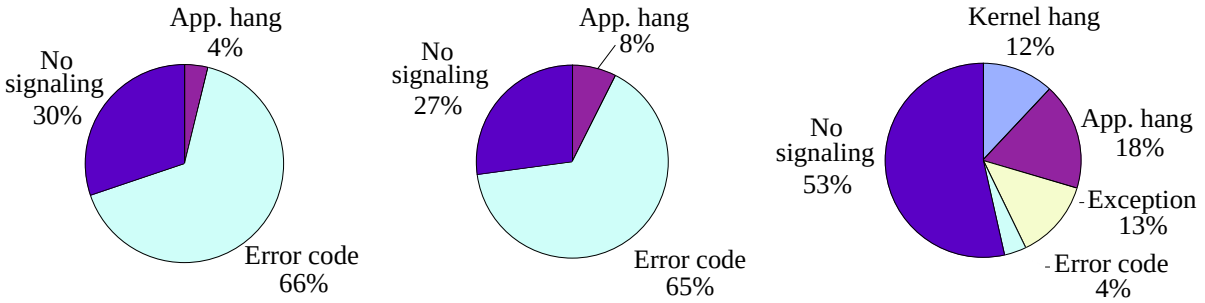
1) Since all internal functions of the kernel are not relevant to our study, the first step consists in choosing the target functions according to the call graph generated for each kernel call. These functions are delimited by inserting comments at the beginning and at the end.
2) The second step consists in inserting, before compilation, blocks of code, called saboteur, at the input point of an elementary function.

We have developed an injection controller module, called **sabotage controller** in Figure 2, to enable faults within the kernel. Although several saboteurs can be inserted, only one is activated per experiment. Each insertion is associated with a flag. The set of flags introduced permits the sabotage controller to control the injections. The **injector** in Figure 2 enables the activation of a fault by issuing an ioctl() to the sabotage controller module.

It is worth noting that such an injection technique is intrusive and can only be applied if the source code is available. But this is not at all a problem in the type of controlled experiments we are conducting here. This kind of injection provide very accurate corruptions [13].

3.3 Observation strategy

The classification of failure modes is a crucial issue to draw out relevant insights. We distinguish five outcomes split into the detection and the non-detection classes. The kernel signals an error by either returning an error code or by handling a processor exception.



a- API invalid parameters (507 experiments) *b- Bit-flip in API parameters (1890 experiments)* *c- Bit-flip in internal function parameters (552 experiments)*

Figure 4- Failure mode rates for the three injection techniques

As the accuracy of the error reports is not the main objective of our study, we do not discriminate the cases where the kernel returns inadequate error code.

The non-detection class encompasses the three following outcomes: i) *kernel hang*, ii) *application hang* and iii) *no signaling*. As we concentrate on the analysis of the operating system reaction, we do not detail application fails. We recall that the idea is to rely on the error detection mechanisms of the kernel to study whether the errors provoked by using different techniques are statistically equivalent. Due to the non-determinism attached to the behavior of the target system and the different faults that are considered, looking for an exact matching of experiments would be irrelevant. Peripheral cards and the scheduling of kernel threads are the major causes of non-determinism. To minimize the system non-determinism, we execute the experiments just after the system ends booting. Also, we try to disable the corresponding drivers and some kernel threads for each campaign, though this is not always possible in practice as this could divert the system from its nominal configuration. A specifically designed tracing tool inserts break points into the kernel to monitor various events (system call trap, interrupt handling, context switch…), and allows us to detect various causes of system non-determinism.

4. Results and analysis

The above framework applies to all kernel functional components. However, in this paper, we focus the analysis on the **scheduling component**. The experimentation has been achieved on version 2.4.0 of the Linux kernel. The developed workload activates the elementary functions associated with this component in a simple way. We have selected six kernel calls to be activated by the scheduling component related workload: i) `setpriority`, `sched_setscheduler` and `wait4` for the process scheduling and ii) `setitimer`, `nanosleep` and `gettimeofday` for the timer management. Other system calls associated with the scheduling component are used by the workload but are not relevant to our study since they have no parameters, such as the `fork` system call.

The workload is composed of three processes: a main one that creates two children. The main process changes its priority (`setpriority`) and creates two other processes before yielding the processor and waiting for the end of the other processes (`wait`). One of them changes its scheduling policy to FIFO (`sched_setscheduler`), while the other one sleeps for 5 ms (`nanosleep`). The sleeping process wakes up and issues various calls (`setitimer`) to update its timers. All the processes issue, along the execution of the workload, the `gettimeofday` call.

The results of the 2949 experiments are given in Figure 4. The dominant observation in the kernel API injection experiments (Figure 4a and 4b) is returning error codes (66% and 65% respectively). This shows the effectiveness of the checks implemented at the Linux kernel API level. Note the difference in the generated failure modes between the injections at the kernel API level and in its internal functions (Figure 4c). There are two additional failure modes that do not appear when injecting at the API level. The error code rate when injecting inside the kernel is low (4%). On the other hand, 13% of faults are detected by hardware-generated exceptions, which means that 17% of faults lead to detected errors. 12% of the injected faults lead to kernel hang.

Let us analyze the reasons of the difference between the injections at the kernel API level and in its internal functions. Generally, the kernel calls in Linux consist in up calls to internal functions as for `sched_setscheduler` in Figure 1 for. It calls one function (`setscheduler`), which fulfills the required service. One may assume that injections at the second depth level of this kind of kernel calls (`sched_setscheduler`, `gettimeofday` and `setitimer`) lead to the same error code. This is true for the `sched_setscheduler` kernel call

where "Invalid Argument" and "Non Existent Process" error codes are generated even when injecting in the third level of the kernel function call graph. However, injections in the second level of the `setitimer` kernel call do not provoke "Bad Address" error code and provoke only an "Invalid Argument" error code. This means that the error detection mechanisms for this function are implemented only in the first level. The analysis of the source code of the underlying function supports this statement. In fact only the value of the first parameter is verified in the underlying elementary function, which explains the presence of the "Invalid Argument" error code alone.

Figures 5a and 5b refine the results of Figures 4a and 4b respectively and show that, for a given kernel call, except for the `nanosleep` case, all error codes generated by the two injection techniques at the API level are of the same nature. Even though, the overall error code rate is almost similar and the generated error codes are the same, they are not always statistically equivalent, except for certain cases such as `setpriority`, `gettimeofday` and `wait4`. Thus, more refinement is needed to derive relevant insights. The dominant error code is "Bad Address". It is present in the same five kernel calls out of six, for the two techniques. These five kernel calls use either a read pointer data type or a write pointer data type.

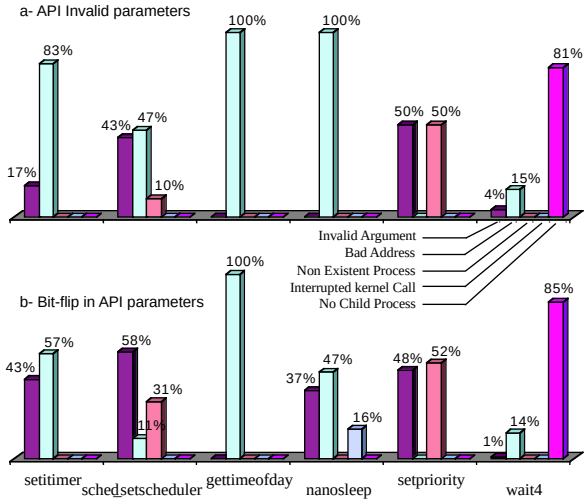


Figure 5- Returned error codes provoked by injection at the API level

A more detailed analysis further supports this statement. Figure 6 provides an example of the type of in depth insight that can be obtained from the experiments. This figure shows that two kinds of error codes ("Non Existent Process" and "Invalid Argument") were observed when flipping bits in the first parameter of the Permission flag class of the `setpriority` kernel call. The "Non Existent Process" error code was provoked by flipping one of the two first bits (0 and 1). The "Invalid Argument" was provoked by flipping any of the remaining bits (2-31). Only two invalid values [*-1* and *ULONG_MAX]* injected in this parameter provoked the "Invalid Argument" and none of them provoked the "Non Existent process" error code. Indeed, the `setpriority` kernel call defines the scheduling priority of either a process, a process group or a user processes. The first parameter defines which scheduling priority will be modified by setting one of these flags: PRIO_PROCESS (0), PRIO_PGRP (1) or PRIO_USER (2), which are coded in the two first bits. The second parameter is interpreted with respect to the value of the first one, thus it could be: a process identifier, a process group identifier or a user identifier for which the scheduling priority of its processes is to be change. The workload uses the `setpriority` kernel call to modify the scheduling priority of one process (PRIO_PROCESS). So, by flipping one of the two first bits of the first parameter (that are the most significant bits), we obtain either PRIO_PGRP or PRIO_USER instead. The second parameter no longer matches the resulting values of the first parameter. Also, it is important to note the impact of the system state. In fact, if the second parameter contains a valid process group after corrupting the value of PRIO_PROCESS to PRIO_PGRP, the kernel will not be able to detect the error, and the application result, if not the entire system state, will surely be corrupted.

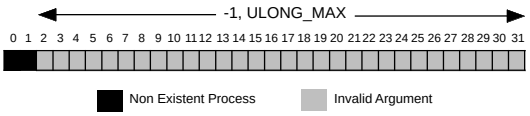


Figure 6- Bit-flip / invalid parameter mapping (setpriority first parameter case)

The above observations further support the view that flipping single bits in kernel call parameters at the API level produces more erroneous behaviors than applying invalid parameters.

5. Conclusion

This work compares the impact of three types of SWIFI techniques. Two of them target the kernel call parameters with two different fault models, namely: i) bit-flip and ii) invalid parameters, and the third one targets the parameters of the kernel calls underlying functions. We have developed an injection environment that supports the three injection techniques. The results presented in this paper target the Linux-kernel scheduling component. They concern six kernel calls invoked by this component.

The comparison of the results relies on typical kernel failure modes (e.g., exceptions and kernel hangs) and on the error detection mechanisms provided by the kernel.

The **bit-flip in internal function injection technique** showed different erroneous behaviors compared to the two other techniques. Many hardware exceptions were triggered by this technique, and the rate of the generated error codes was lower than for the other techniques. This tends to indicate that it is unlikely that internal software faults (residual design faults or device drivers caused faults) could be easily emulated by injecting only at the API level, at least for the Linux kernel. Further work is in progress to better study this issue.

The **bit-flip in kernel call parameter injection technique** is an easy task and does not need any a priori analysis of the parameter data types. However, it requires a lot of time, as it needs 32 injections per parameter.

The **invalid parameter injection technique** takes less time for a complete campaign compared to a complete bit-flip one. But is a difficult issue, since it needs a priori analysis, though this analysis could be done only once such as the Ballista based POSIX test suite, which could be applied to all the POSIX compliant systems. From the efficiency point of view, the single bit-flip injections provoked more erroneous behaviors than the invalid parameter injection technique. We presented a detailed case where the invalid parameter injections were unable to reproduce an error code that was provoked by bit-flips. This might indicates that we have to enrich the set of invalid parameters.

We plan to implement more error detection mechanisms such as assertions into the kernel to enrich the erroneous behavior observation and to obtain more detailed traces allowing analysis of error propagation channels. Finally, we intend to analyze errors produced by real faults, already activated in Linux, that have been published. Our ultimate aim is to compare the set of errors produced by injected faults to the set of errors produced by real faults to identify a set of representative faults to be injected in order to characterize the OS behavior in presence of faults, i.e., benchmark the OS dependability.

Acknowledgment This work is partially supported by the European Community (Project IST-2000-25425 DBench: Dependability Benchmarking). The work presented in this paper has largely benefited from many fruitful discussions with Jean-Claude Laprie from LAAS. Also, we would like to thank Benjamin Lussier, Moslem Belkhiria and Thomas Marteau who contributed to the experiments during their training period at LAAS.

References

[1] J. V. Carreira, D. Costa and J. G. Silva, "Fault Injection Spot-checks Computer System Dependability", *IEEE Spectrum*, vol. 36, pp. 50-55, 1999.

[2] D. T. Stott, G. Ries, M.-C. Hsueh and R. K. Iyer, "Dependability Analysis of a High-Speed Network Using Software-Implemented Fault Injection and Simulated Fault Injection", *IEEE Trans. on Computers*, vol. 47, no. 1, pp. 108-119, 1998.

[3] E. Fuchs, "Validating the Fail-Silence of the MARS Architecture", in *Proc. DCCA-6, Grainau, Germany*, 1998, pp. 225-247.

[4] H. Madeira, D. Costa and M. Vieira, "On the Emulation of Software Faults by Software Fault Injection", in *Proc. DSN-2000*, New York, NY, USA, 2000, pp. 417-426.

[5] J.-C. Fabre, F. Salles, M. Rodríguez Moreno and J. Arlat, "Assessment of COTS Microkernels by Fault Injection", in *Proc. DCCA-7, San Jose, CA, USA*, 1999, pp. 25-44.

[6] P. Koopman and J. DeVale, "Comparing the Robustness of POSIX Operating Systems", in *Proc. FTCS-29*, Madison, WI, USA, 1999, pp. 30-37.

[7] J. E. Foster and B. P. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing", in *Proc. 4th USENIX Windows System Symposium*, Seattle, WA, USA, 2000.

[8] I. T. Bowman, R. C. Holt and N. V. Brewster, "Linux as a Case Study: Its Extracted Software Architecture", in *Proc. 21st Int. Conf. on Software Engineering*, Los Angeles, CA, USA, 1999.

[9] R. Chillarege & al., "Orthogonal Defect Classification - A Concept for In-Process Measurements", *IEEE Trans. on Software Engineering*, vol. 18, no. 11, pp. 943-956, 1992.

[10] *http://www.wi.leidenuniv.nl/~wichert/strace*.

[11] A. D. Alexandrov, M. Ibel, K. E. Schauser and C. J. Scheiman, "Extending the Operating System at the User Level: the Ufo Global File System", in *Proc. USENIX*, Anaheim, CA, USA, 1997, pp. 77-90.

[12] M. Rodríguez, F. Salles, J.-C. Fabre and J. Arlat, "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid", in *Proc. EDCC-3*, Prague, Czech Republic, 1999, pp. 143-160.

[13] J. Christmansson, M. Hiller and M. Rimén, "An Experimental Comparison of Fault and Error Injection", in *Proc. ISSRE'98*, Paderborn, Germany, 1998, pp. 369-378.