Analysis of the Effects of Real and Injected Software Faults: Linux as a Case Study*

Tahar Jarboui, Jean Arlat, Yves Crouzet, Karama Kanoun, Thomas Marteau

To Appear in Proc. 9th Pacific Rim International Symposium on Dependable Computing (PRDC-2002)
Tsukuba, Japan, December 16-18, 2002 (IEEE CS Press)

Analysis of the Effects of Real and Injected Software Faults: Linux as a Case Study*

Tahar Jarboui, Jean Arlat, Yves Crouzet, Karama Kanoun and Thomas Marteau
LAAS-CNRS
7 avenue du Colonel Roche
31077 Toulouse Cedex 4 — France

Abstract

The application of fault injection in the context of dependability benchmarking is far from being straightforward. One decisive issue to be addressed is to what extent injected faults are representative of actual faults. This paper proposes an approach to analyze the effects of real and injected faults.

1. Introduction

Fault injection has long been recognized as a pragmatic means to assess the dependability of computer systems [1], in some cases in complement with other more formal approaches (e.g., see [2-4]). Numerous techniques and tools were proposed [5, 6] that have been widely applied both in research projects and industry.

Nevertheless, the plausibility of the fault models supported with respect to real faults is a concern that has often been related to fault injection-based experiments, even when they were targeted to the evaluation of a specific computer architecture.

Such a concern is indeed even more acute in the case of dependability benchmarking, due to the fact that the ultimate objective is to compare alternative systems. Accordingly, among the various attributes (such as workload, *faultload*, measurements and measures) that have been introduced within the general framework defined by the DBench project [7] to characterize a dependability benchmark, the determination of a *representative faultload* has been identified as one of the key challenges.

Based on the fact that similar errors can be induced by different types of faults, we advocate that what actually matters is not to establish a correspondence in the fault domain, but rather in the error domain, i.e., among the consequences that are provoked by the application of a set of real faults or a set of injected faults. More precisely, two main viewpoints and related questions have to be considered:

1) Fault equivalence (between fault injection techniques): To what extent do distinct fault injection techniques lead to similar effects (errors and failures) ?
2) Fault representativeness (of a given fault injection technique): To what extent are the effects of injected faults similar to those provoked by real faults or by a representative fault model?

Concerning the types of systems or components that can be considered as target systems, operating systems are definitively privileged candidates for dependability benchmarking. Indeed, any malfunction of the OS may have a strong impact on the dependability of the global system. Linux has been mainly used as the target OS to carry out this study because of its open source status that significantly facilitates the supporting of the internal controllability and observability capabilities required by the experiments to be conducted. Nevertheless, it is worth noting that Linux is now considered a “real world” OS that is being included in a wide range of applications including those with dependability requirements.

This work proposes a comprehensive strategy to: i) study the equivalence between the fault injection techniques at the API level, ii) compare the effects of API injected faults with internal injected faults, and iii) analyse the representativeness of the injected faults with respect to real faults.

We have developed an experimental framework that support three injection techniques. It is well detailed in [8].

The paper is structured as follows. Section 2 presents the considered set of faults. Section 3 describes the observation levels. Some of the results obtained are then presented and discussed in Section 4. Finally, Section 5 concludes the paper.

2. Considered software faults

Many fault representativeness studies targeted the physical faults and they all agreed on the fact that the bit-flip is a representative fault model of physical faults (e.g., see [9]). However, besides a few studies [10, 11], less attention has been paid to software faults, which are

* This work is partially supported by the European Commission: project DBench - Dependability Benchmarking (IST-2000-25425)

considered as the first causes of actual system outages [12].

2.1. Real software faults in Linux

The study of the real faults that are observed in the OS kernel source code permits a better understanding of the erroneous behaviors that are provoked in reality. One way to collect such information is the analysis of the change logs provided with each new kernel version. They include comments about the various fixes and additions applied to the previous version. However, it is not always easy to associate a comment with its corresponding fix.

A meta-compiler, developed recently at the Stanford University [13], permits the detection of real faults in OS kernels. It is based on system-specific checkers defined after a static analysis of the source code. The goal of a checker is to verify programming rules inside the kernel. The faults revealed by these checkers are published on the web.

The types of real faults that we will consider are those revealed by the BLOCK and the NULL checkers. The outcomes that are likely to be induced by these faults are respectively "Kernel hang" and "Exception". The latter is usually followed by a "kernel panic" mode. They correspond respectively to 42% and 25% of the faults revealed by the considered checkers. The analysis of their eventual provoked errors allows us to develop some assertions explained more in detail in Section 3.2.

2.2. Injected software faults

We present hereafter the considered model for the software system from which we derive the injected fault models. We distinguish between external and internal faults.

2.2.1 Functional decomposition

Based on the work presented in [14], 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. This functional decomposition of Linux, which is a monolithic operating system, is only used to facilitate the analysis. 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, building on work presented in [15], we were able to generate, at kernel compilation, a call graph for each kernel call. A call graph identifies 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`. It 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.

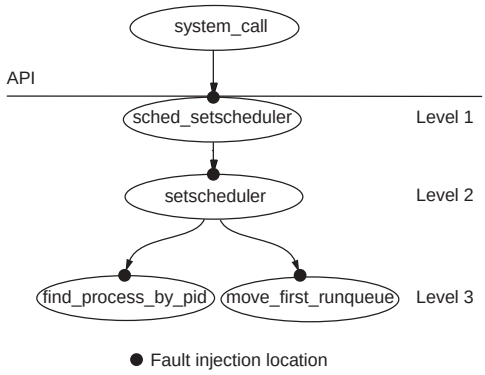

Figure 1. sched_setscheduler Call graph

In this paper, three fault models are considered. 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. 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*).

Another important fault parameter is the trigger condition of the fault. The trigger condition is the event that leads to the injection of the fault. The considered trigger condition is the interception of the targeted system call. The fault is thus injected depending on the selected level (external or internal).

2.2.2. External faults

External faults mimic faults from the application level and they test the robustness of the kernel.

The bit-flip is a generic fault model. It primarily simulates hardware faults. The goal of the proposed comparison is to determine to what extent it can simulate the invalid argument injection technique, which is a more focused fault model for software faults.

We compare error sets caused by bit-flips (as for MAFALDA [16]) into system call parameters with those caused by invalid parameters (as for Ballista [17]). The parameter values are corrupted by i) issuing exhaustive bit-flips (32 per parameter) or, ii) replacing them with invalid values. Based on the work related to Ballista, and especially its online demonstration site, eight classes of invalid parameters are defined.

2.2.3. Internal faults

Internal faults emulate various classes of faults such as those defined in the Orthogonal Defect Classification [18]

(e.g., assignment, checking, interface, etc). We consider only the interface class. The set of experiments targets the kernel internal functions that are not reachable via the API.

3. Observation levels

The main objectives of the conducted experiments are to:

- Study the possible equivalence between the fault injection techniques at the API level,
- Compare the effects of API injected faults with internal injected faults,
- Analyze the representativeness of the injected faults with respect to real faults, based on the nature of the produced errors.

The comparison between two fault models is achieved through the comparison of their respective effects. The highest abstraction level consists in the quantification of the observed failure modes. If the observed failure modes, after the injection of two different fault models, are different we can state that these fault models are not equivalent. However, if the failure modes are similar, a refinement of the observations might be needed to be able to provide more affirmative conclusions. The first refinement we are considering consists in taking into account the error detection mechanisms built-in within the kernel to enhance the observability capabilities. The second refinement to further enhance the observability consists in implementing extra internal assertions within the kernel.

3.1. Experiment outcomes

We distinguish two main types of outcomes as the result of a fault injection experiment: *reported* and *non-reported failures*. For sake of conciseness, both types will be considered as *failure modes*. The considered outcomes are detailed hereafter.

3.1.1. Reported failures

The reported failures correspond to hardware raised exception or to error codes returned by the kernel:

- A hardware exception is raised. If the exception is raised in user mode, the kernel sends a signal to the process that caused the exception. However, if it is raised when running in kernel mode, either the running process is killed or the kernel enters the "panic" mode.
- An error code is returned. As the accuracy of the error reports is not the main objective of our study, we do not discriminate the cases when an incorrect error code is returned (also termed "hindering" in [19]).

3.1.2. Non-reported failures

The non reported failures correspond to either a kernel or an application hang:

- The kernel hangs. A kernel hang can be observed due to an infinite loop within the kernel or when it is waiting for an event that never occurs while interrupts are disabled.
- An application hangs. This can be due to an infinite loop that the application executes or it can be waiting on a kernel wait queue for an event that will never occur. In the latter case, the application does either accept signals, in which case we can force it to quit, or it does not accept signals, in which case we need to reboot the kernel.

When none of the previous events is observed, then a "*no signaling"* outcome is assumed.

3.2. Internal assertions

This section presents the additional observation mechanisms that permit a finer tracing of the errors provoked by the injected faults.

We define a kernel control flow as the sequence of instructions carried out by the kernel following a kernel call, an interruption or an exception. The code of Linux is reentrant, i.e., several flows of control are carried out in parallel.

We implement checkpoints by means of assertions located at the level of internal functions. A checkpoint can belong to several kernel control flows. The combination of these values determines to some extent the consistency of the kernel-state.

We have developed two types of assertions. The first type is based on the analysis of propagation paths of some real faults presented in Section 2.1. The second type consists in implementing extra observation and error detection mechanisms within the kernel. Such mechanisms could easily be implemented by kernel developers, but, they are seldom included, essentially for performance reasons.

3.2.1. Assertions based on real faults

The Linux real faults database presented in Section 2.1 has shown that BLOCK and NULL fault classes are the most frequent in the Linux kernel. They are most of the time located in the Linux device drivers. For Linux developers, while some errors can cause the kernel to enter an endless loop and thus leading it to hang, most errors manifest either as null pointer dereferences or by the use of other incorrect pointer values (the usual outcome of such errors is an "oops" message). As a consequence, we put emphasis on BLOCK and NULL classes of faults.

We have analyzed the propagation of the effects induced by these faults. We identified the errors that are likely to be provoked by these faults at the kernel level. Then we

implemented assertions that were able to detect such errors.

BLOCK faults correspond to calling blocking functions when for example interrupts are disabled. We identified all the internal functions present in the Stanford real fault database (see Section 2.1) that are considered as blocking functions. We implemented assertions at 8 of these functions entry points to check whether interrupts are disabled or not.

3.2.2. Other specific assertions

The development of this type of assertions is based on the analysis of internal functions. They are inserted at the entry and exit points of internal functions, part of the call graph of a given kernel call.

As an example, the internal function `find_task_by_pid` that is present in the call graph of the `sched_setscheduler` kernel call, takes a process identifier as an input parameter and returns a pointer to the structure that characterizes the process. A simple assertion is to verify that the process identifier associated to the returned structure is correct, i.e., equal to the input parameter.

In addition to these extra detection mechanisms, we implement assertions that monitor global kernel variables indicating the global kernel-state. They give us an internal view of the whole impact of the injected faults. We select indicators for each functional component. The memory pressure for example reveals the capacity of the kernel to serve user applications in term of memory allocation. Thus, in the main function of Linux dealing with memory allocation, we placed an assertion that gives us the value of such variable in order to post-analyze the evolution of memory pressure during an experience.

3.3. Implementation

The experimental set-up is composed of a target machine (Pentium III PC running Linux 2.4.0) on which the faults are injected and a host machine (SUN workstation) that manages the experiments. The execution trace that contains the assertions and the detection modes are stored on the target machine. Whereas, the failure modes such as the kernel or the application hang are stored on the host machine. These two sets form the results.

A module is inserted into the target kernel to retrieve the execution trace. The checkpoints along with their associated assertions are inserted within the kernel before compilation. The execution trace is saved on a file and analyzed afterward.

We recall that the reported failures include returned error codes and exceptions. We have identified from the source code 20 error codes for the scheduling and the memory components. These outcomes are deduced from the analysis of the application and the kernel execution traces.

The host machine is able to notify the non-reported failures, i.e., the kernel and application hangs. The target machine signals to the host machine the beginning and the end of an experiment. After an experiment start, the host machine waits two minutes. If the target machine doesn't signal an experiment end within this interval, the host tries to establish a connection with the target. If it succeeds, we assume an application hang, and the host reboots the target automatically. Else, we assume a kernel hang, in which case the target needs to be rebooted manually. When the observed behavior does not match any of the previous outcomes, a "no signaling" is reported.

4. Results

The conducted experiments addressed the scheduling and the memory components. We carried out 4470 experiments. Bit-flips in internal function parameters target 340 internal functions. These internal functions belong to the call graphs of six scheduling kernel calls and four memory kernel calls, target of the external injections. Table 1 summarizes the experiment distribution per functional component.

Table 1 Experiment distribution

	Invalid argument	Bit-flip in API parameters	Bit-flip in internal function parameters
Scheduling	507	1890	552
Memory	101	1019	401
Total	608	2909	953

The number of experiments when injecting invalid values is less than when injecting bit-flips. In fact, between 3 and 6 invalid values are associated to each kernel call parameter for the former injection technique, while 32 bit-flips are issued to each kernel call parameter for the latter. That's why the injection of invalid parameters is eight times faster. However, it requires more time for preparation.

4.1. Analysis of the failure modes

In the following we propose three kinds of analyses. The first one will permit us to compare the provoked errors of the external fault models. Then we intend to compare the errors provoked by all the injected fault models. Finally, we present details about the influence of the target kernel functional component. The failure modes are given in Figure 2.

The two external fault models provoke approximately the same failure modes in terms of nature and quantity. The dominant failure mode (Figure 2-a and 2-b) is returning error codes (57% and 45% respectively). This shows the effectiveness of the checks implemented at the Linux kernel API level. A detailed analysis of the nature of the returned codes is presented in Section 4.2.1.

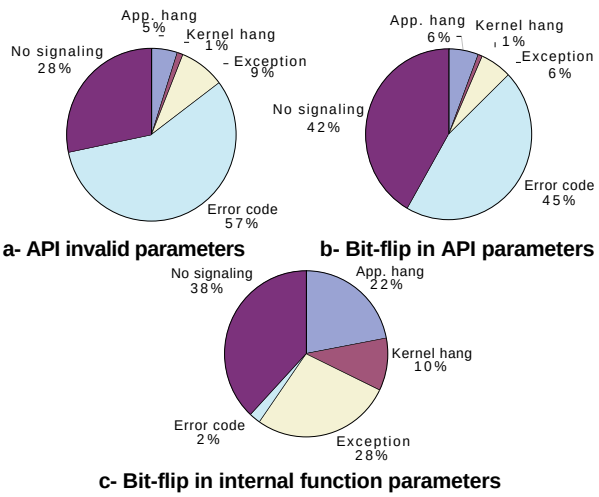

Figure 2. Failure mode distributions

API invalid parameters provoke less "no signaling" cases. Such an outcome is unusable and cannot be interpreted. However, it could lead to a failure in a different context than the one being considered in these experiments.

With respect to all injected faults, we notice in Figure 2 the difference in the generated failure modes between the injections at the kernel API level (Figure 2-a and 2-b) and in its internal functions (Figure 2-c). The error code rate when injecting inside the kernel is low (2%). On the other hand, 28% of faults have been detected by hardware-generated exceptions, which means that 30% of faults have lead to detected errors.

Let us analyze the reasons for 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 illustrated in Figure 1. The latter 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 at 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 checked in the underlying elementary function, which explains the presence of the "Invalid argument" error code alone in some experiments.

Figure 2-c shows that 10% of the injected faults in internal function parameters, leads to kernel hang, which is significantly higher than the 1% observed when injecting external faults. Also, 22% of the faults injected in internal function parameters, leads to application hang, which is four times more than the observed percentage of external faults (5% and 6%).

4.2. Analysis of the returned error codes

Based on the returned error codes, two kinds of analysis can be carried out. The first consists of the analysis of the nature of error codes, and the second consists of the analysis of whether other subsequent kernel calls return error codes.

4.2.1. Error codes provoked

Figure 2-a and 2-b show that the rate of error codes is greater when injecting invalid arguments than when injecting bit-flips. Figure 3-a and 3-b refine these results and show that generally**,** for a given kernel call, all error codes generated by the two injection techniques at the API level are of the same nature. Yet, we notice a slight advantage for the bit-flip injection technique that provides more error codes than the invalid parameters. Also, the rate associated with each error code is not always equivalent, except for certain cases such as wait4.

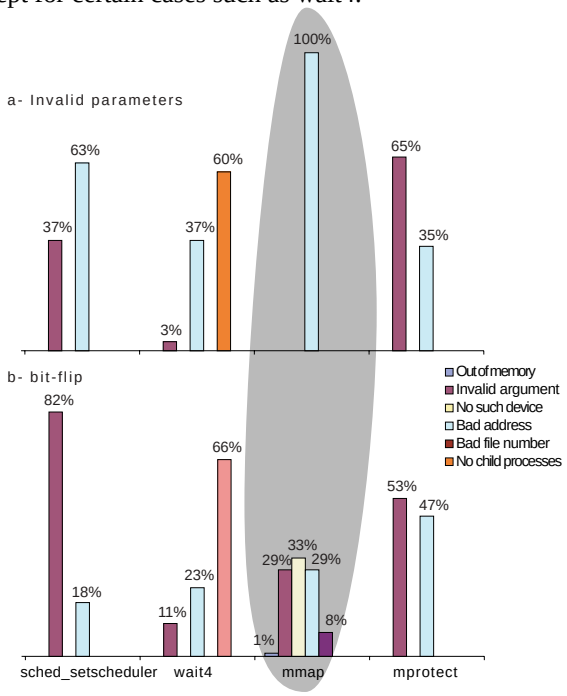

Figure 3. Returned error code analysis

The `mmap` kernel call provides a singular behavior. As illustrated in Figure 3, five error codes were observed when injecting bit flips ("Out of memory", "Invalid argument", "No such device", "Bad address", "Bad file number"), while

only one error code is returned when injecting invalid parameters ("Bad address").

4.2.2. Error propagation

When injecting within the kernel calls of the memory component issued by the memory workload, we observed error propagation to other functional components. We did not notice such propagation for the scheduling component.

Figure 4 illustrates error propagation rates when injecting faults in `brk`, `mprotect` and `munmap` kernel calls. The `mmap` kernel call doesn't provoke such propagation when injecting either invalid parameters or bit-flips. For each arc, the first percentage is associated with invalid argument technique and the second is associated with the bit-flip technique. We take into account all cases where an error code is observed. Error codes are returned either by `open` (file system component) or by both `open` and `wait4` (scheduling component). For example, corrupting the `brk` kernel call with invalid argument lead in 30% of the cases to `open` returning an error code and in 55% of the cases to both `open` and `wait4` returning error code. That does not mean that in 15% of the cases an error code is returned by `brk`. Figure 4 illustrates only the error propagation cases.

Injecting invalid parameters promotes error propagation for the three kernel calls among the four considered memory kernel calls. However, injecting bit-flips propagates errors only in the case of `mprotect`, with rates equivalent to those of invalid parameter technique (7% versus 9% propagate to `open` and 87% versus 64% propagate to both `open` and `wait4`).

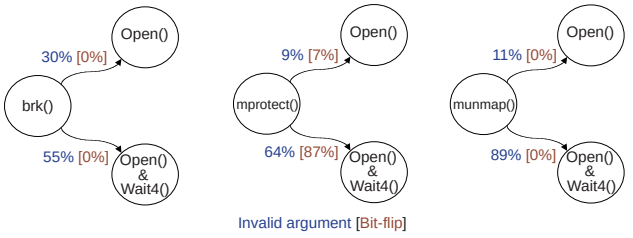

Figure 4. Error propagation

4.3. Assertion analysis

We detail hereafter the observations made thanks to the implemented assertions. We recall that these assertions are implemented after the analysis of the source code. We are considering successively the two assertions that lead to relevant results.

The first assertion reports the `brk` kernel call activity. We have observed in normal operation (in the absence of faults) that the size of the memory data segment has not to be changed in 64% of the cases (`brk` doesn't carry out a specific treatment). Figure 5 illustrates the cases where we noticed a deviation from this normal invocation. Injected faults in the internal functions used by the `mmap` kernel call for example improve this rate by 0.5%, which is not the case for the bit-flip and the invalid parameter techniques at the API where the rate decrements respectively by 1.5% and 2.5%. This rate decreases by 7%, which is the worst case, when injecting faults in the parameters of the internal functions called by the `munmap` kernel call.

No such deviation has been observed when we inject faults in the kernel calls of the scheduling component. This is likely due to the fact that there is no error propagation from the scheduling component to the memory component.

The second assertion concerns the "memory pressure". It represents the number of allocation requests that the kernel is trying to satisfy and thus the current memory load. The greater this value, the more the memory becomes a critical issue. Figure 6 presents the percentages of experiments where the memory pressure increases by more than 50% after a fault injection in the memory kernel calls.

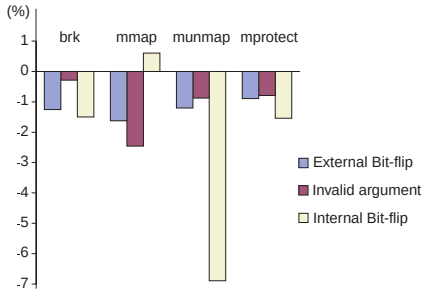

Figure 5. `brk` failure rate variations

The average value of memory pressure is calculated before and after a fault injection. No such behavior has been observed in the experiments targeting the scheduling component.

The influence of injecting faults in internal function parameters on the "memory pressure" is constant. For all the memory kernel calls, about 90% of the experiments lead to a memory pressure increase.

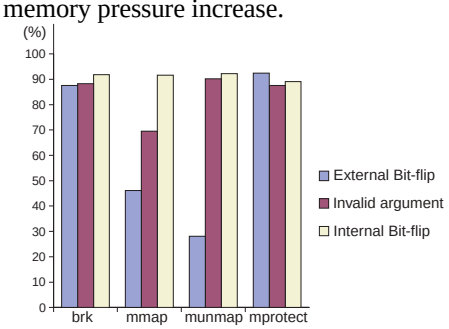

Figure 6. Memory pressure variation

For the external faults, we remark an advantage when injecting invalid parameters since they tend to stress the kernel in more cases than the bit-flip technique. The cases where the bit-flip technique is equivalent to the invalid

parameter technique is when targeting the `mprotect` and the `brk` kernel calls.

4.4. Real *vs.* injected faults

In this section we compare the consequences of the considered real faults (those revealed by the BLOCK and the NULL checkers) and of all the injected faults.

The first observations are based on the failure modes. Let us consider the BLOCK category of real faults. They represent 42% of real faults and lead to "Kernel hang". However, as presented in Figure 2-a and Figure 2-b, only 1% of the external faults, lead to "Kernel hang", while 10% of the internal faults provoke a "Kernel hang".

The second category of real faults that we have considered concerns NULL faults. They constitute 25% of the real faults observed in version 2.4.0 of the Linux kernel. This kind of faults leads to an "Exception". Only 9% and 6% of the external faults lead to an "Exception", while 28% of internal faults provoke an "Exception".

In addition to failure modes, the implemented assertions give an additional observation view point. Their goal is to observe the possible correlation between real errors (caused by real faults) and errors provoked by the three considered injection techniques. The source code inserted assertions are designed to detect the targeted real errors. However, we were not able to activate these assertions with the three injection techniques.

This is due to the difference in the contexts in which the faults are activated. The real faults are revealed in the device drivers. Even though kernel calls and device drivers share some of the kernel internal functions, they are not activated in the same manner, i.e., the injected faults in the parameters of these internal functions are not activated in the same context. This is supported by a specific experiment in which we were able to activate a real fault based assertion. The experiment consisted in injecting faults in the parameters of internal functions called by the device driver functions. The considered workload inserted the network card device driver into the kernel. Although, the activated assertion is designed to detect errors provoked by the faults revealed by the BLOCK checker, no kernel hang was observed. Accordingly, we can conclude that the error remains hidden.

Further work is on going to better assess the representativeness of injected faults with respect to real faults affecting device drivers.

5. Summary and conclusion

This work compares the impact of three types of SWIFI techniques on the Linux OS (version 2.4.0). Two of them target the kernel call parameters at the API (external faults) with two different fault models, namely: i) bit-flip corruption and ii) provision of invalid parameters. The third one applies bit-flips targeting the parameters of the internal functions of the kernel.

In addition to the observed failure modes, either explicitly reported (e.g., exceptions) or not (e.g., hangs), specific assertions were implemented to provide a finer grain reporting; in particular, some of these assertions were deduced from the analysis of the effects caused by real faults.

API-level fault injection is good candidate to assess kernel robustness. Flipping bits in kernel call parameters is easy to implement 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 for a 32-bit kernel and simple data types. Applying invalid parameters is eight times faster, for a complete campaign compared to a bit-flip campaign. But it needs an *a priori* analysis of the kernel call parameters.

Although the provoked failure modes are comparable for both techniques independently from the functional component, the bit-flip injection technique provokes a larger range of error codes than the invalid parameter injection technique. In particular, we have detailed the case of a kernel call (`mmap`) where out of the five error codes provoked by bit-flips only one was provoked by the application of invalid parameters. Nevertheless, applying invalid parameters is more prone than flipping bits to propagate errors, especially from the memory component to other kernel functional components as previously noted. Also, the proportion of experiments that lead to an increase of the memory pressure is more important when injecting invalid parameters.

Table 2 summarizes the pros and cons for each technique. It shows that the invalid parameter technique provides more advantages than the bit-flip technique. In addition, it is worth noting that the *a priori* analysis could be done only once, as is the case for the Ballista-based POSIX test suite, which can be applied to all POSIX compliant systems.

Table 2 Bit-flip vs invalid arguments

	Experiment duration	Ease of application	Error codes provoked	Error propagation	Memory pressure	Significance of experiments
bit-flip	–	+	+	–	–	–
Invalid argument	+	–	–	+	+	+

Compared to the effects induced by external faults, flipping bits in internal function parameters provoked distinct erroneous behaviors. Indeed, many hardware exceptions were triggered by this technique, and the proportion of error codes observed was lower than for the other two techniques. The implemented assertions exhibit further such behavioral differences.

Concerning the representativeness viewpoint, we observed that external faults provoked very distinct behaviors compared to those induced by the real internal faults we considered (device driver faults). In particular, external faults were not able to activate the assertions based on real faults. This tends to indicate that it is unlikely that device driver faults could be easily emulated by injecting only at the API level, at least for the Linux kernel.

The workloads used were selected to activate the kernel functional components in a typical way. The targeted kernel calls we have considered in this work are the most used in practice. However it would be interesting to target additional kernel calls for a each functional component.

Even though the two experimented kernel functional components are judged to be the most critical components, more work is needed to target the other kernel functional components.

Acknowledgment

This work has largely benefited from many fruitful discussions with Jean-Claude Laprie from LAAS. Also, we would like to thank Moslem Belkhiria and Benjamin Lussier who contributed to the experiments during their training period at LAAS.

References

[1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins and D. Powell, "Fault Injection for Dependability Validation — A Methodology and Some Applications", *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166-182, February 1990.

[2] S. Ayache, P. Humbert, E. Conquet, C. Rodriguez, J. Sifakis and R. Gerlich, "Formal Methods for the Validation of Fault Tolerance in Autonomous Spacecraft", in *Proc. 26th Int. Symp. on Fault-Tolerant Computing (FTCS-26)*, Sendai, Japan, 1996, pp. 353-357 (IEEE Computer Society Press).

[3] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie and D. Powell, "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems", *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 913-923, August 1993.

[4] A. Arazo and Y. Crouzet, "Formal Guides for Experimentally Verifying Complex Software-Implemented Fault Tolerance Mechanisms", in *Proc. 7th Int. Conference on Engineering of Complex Computer Systems (ICECCS'2001)*, Skövde, Sweden, 2001, pp. 69-79 (IEEE CS Press).

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

[6] M.-C. Hsueh, T. K. Tsai and R. K. Iyer, "Fault Injection Techniques and Tools", *Computer*, vol. 30, no. 4, pp. 75-82, April 1997.

[7] H. Madeira, K. Kanoun, J. Arlat, Y. Crouzet, A. Johanson and R. Lindström, "Preliminary Dependability Benchmark Framework", DBench Project IST 2000-25425 Deliverable. CF2, Available at http://www.laas.fr/dbench/delivrables.html, 2001.

[8] T. Jarboui, J. Arlat, Y. Crouzet and K. Kanoun, "Experimental Analysis of the Errors Induced into Linux by Three Fault Injection Techniques", in *Proc. Int. Conf. on Dependable Systems and Networks (DSN-2002)*, Washington, DC, USA, 2002 (IEEE CS Press).

[9] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda and M. Violante, "Experimentally Evaluating an Automatic Approach for Generating Safety-Critical Software with respect to Transient Errors", *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2231-2236, December 2000.

[10] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults-Based on Field Data", in *26th Int. Symp. on Fault Tolerant Computing*, Sendai, Japan, 1996.

[11] H. Madeira, D. Costa and M. Vieira, "On the Emulation of Software Faults by Software Fault Injection", in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2000)*, New York, NY, USA, 2000, pp. 417-426 (IEEE CS Press).

[12] I. Lee and R. K. Iyer, "Software Dependability in the Tandem GUARDIAN System", *IEEE Transactions of Software Engineering*, vol. 21, no. 5, pp. 455-467, 1995.

[13] D. Engler, B. Chelf, A. Chou and S. Hallem, "Checking System Rules Using System-Specific Programmer-Written Compiler Extensions", in *Proc. 4th Symp. on Operating Systems Design and Implementation (OSDI-2000)*, San Diego, CA, USA, 2000 (USENIX).

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

[15] M. Devera, *Devik's MM Gallery* http://luxik.cdi.cz/~devik/mm.htm, 2001.

[16] M. Rodríguez, F. Salles, J.-C. Fabre and J. Arlat, "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid", in *Proc. 3rd European Dependable Computing Conf. (EDCC-3)*, Prague, Czech Republic, 1999, pp. 143-160 (Springer).

[17] P. Koopman and J. DeVale, "Comparing the Robustness of POSIX Operating Systems", in *Proc. 29th Int. Symp. on Fault-Tolerant Computing (FTCS-29)*, Madison, WI, USA, 1999, pp. 30-37 (IEEE CS Press).

[18] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray and M. Y. Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurements", *IEEE Transactions of software engineering*, vol. 18, no. 11, pp. 943-956, November 1992.

[19] P. J. Koopman, J. Sung, C. Dingman, D. P. Siewiorek and T. Marz, "Comparing Operating Systems using Robustness Benchmarks", in *Proc. 16th Int. Symp. on Reliable Distributed Systems (SRDS-16)*, Durham, NC, USA, 1997, pp. 72-79 (IEEE Computer Society Press).