PAPER Special Issue on Dependable Computing

Impact of Internal and External Software Faults on the Linux Kernel*

Tahar JARBOUI†, Jean ARLAT†, Yves CROUZET†, Karama KANOUN†, *and* Thomas MARTEAU†, *Nonmembers*

SUMMARY 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 the considered faults. This paper proposes an approach to analyze the effects of real and injected faults.

key words: *software faults, error analysis, fault injection, operating system, dependability benchmark*

1. Introduction

Fault injection has long been recognized as a pragmatic means to assess the dependability of computer systems [1], in some cases in conjunction with other more formal approaches (e.g., see [2]–[4]). Numerous techniques and tools have been proposed [5], [6] and 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 raised regarding fault injection-based experiments, even when they were targeted to the evaluation of a specific computer architecture [7].

These concerns are 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 to characterize a dependability benchmark [8], the determination of a representative faultload has been identified as one of the key challenges.

The concepts and terminology used to describe fault pathology are presented in [9]. A system may fail either because it does not comply with the specification, or because the specification did not adequately describe its function. An error is that part of the system state that may cause a subsequent failure: a failure occurs when an error reaches the service interface and alters the service. A fault is the adjudged or hypothesized cause of an error. A fault is active when it produces an error; otherwise it is dormant.

Based on the fact that similar errors can be induced by different types of faults, we believe 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 different 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 for dependability benchmarking, operating systems are privileged candidates. Indeed, any malfunction of the OS may have a strong impact on the dependability of the global system. Linux has been used as the target OS to carry out this study because the availability of its source code facilitates the implementation of internal controllability and observability capabilities, required by the experiments to be conducted. It is worth noting that Linux is now considered an industrial-strength OS that is being incorporated in a wide range of applications, including those with dependability requirements.

In this paper, we present and implement a strategy to: i) study the equivalence between the fault injection techniques at the API (Application Programming Interface) level, ii) compare the effects of API injected faults and internal injected faults, and iii) analyse the representativeness of the injected faults with respect to real faults. The implementation is based on the experimental framework presented in [10].

The paper is structured as follows. Section 2 presents the proposed strategy. Section 3 details the considered set of faults. Section 4 describes the observation levels. Some of the results obtained, from experiments targeting version 2.4.0 of the Linux kernel, are presented and discussed in Sect. 5. Finally, Sect. 6 concludes the paper.

Manuscript received April 7, 2003.
Manuscript revised July 7, 2003.

†The authors are with LAAS-CNRS, 7 avenue du Colonel Roche, 31077 Toulouse, France.

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

2. Proposed Strategy

Figure 1 presents the proposed strategy to study fault representativeness and fault equivalence. We identify several relevant levels of a computer system where faults can occur and errors can be observed (e.g. hardware, kernel, application, operation). Concerning faults, these levels may correspond to levels where real faults (reference level) are considered and (artificial) faults can be injected. Error detection mechanisms provide convenient built-in monitors for the errors being produced. For characterizing the behavior of a computer system in presence of faults, it is not necessary that the injected faults be close to the target faults (reference); it is sufficient that they induce similar behaviors. Indeed, similar errors can be induced by different types of faults (e.g., a bit-flip in a register or memory cell can be provoked by an heavy-ion or as the result of a glitch provoked by a software fault). What is important is not to establish an equivalence in the fault domain, but rather in the error domain.

What matters is that the respective error propagation paths converge before the level where the behaviors are observed. Two important parameters can be defined on these various levels:

- the distance *dr* that separates the level where faults are injected from the reference fault level(s);
- the distance *do* that separates the level where the faults are injected from the levels where their effects are observed.

Decreasing *dr* and increasing *do* makes it more likely that the injected faults will exhibit behaviors similar to those provoked by the targeted reference faults.

Many fault representativeness studies targeted physical faults and agreed on the fact that the bit-flip is a representative fault model of physical faults [11]. However, besides a few studies [12], [13], less attention has been paid to software faults, which are reported as the primary causes of actual system outages [14]. We focus in this work on software faults that might affect the kernel, in a dependability benchmarking context.

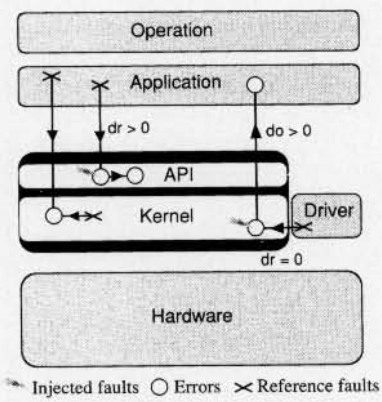

Fig. 1 Fault representativeness and equivalence strategy.

The device drivers and the API are considered at the same level as the kernel. Some real faults have been identified in the Linux device drivers. We inject faults inside the kernel, where the *dr* distance is zero. These experiments are interesting to study error propagation channels, but are not useful for dependability benchmarking. We also inject faults at the API level. These faults simulate application level faults, i.e. at a non-zero distance *dr*. The observations are made at the kernel and the application levels.

3. Considered Faults

This section details the real software faults in Linux and the injected software faults.

3.1 Real Software Faults in Linux

The study of real faults that are observed in the OS kernel source code permits a better understanding of the erroneous behaviors that are actually provoked.

A meta-compiler, developed recently at Stanford University [15], permits the detection of real faults in OS kernels. It is based on system-specific checkers 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 (http://metacomp.stanford.edu/linux/list.php3).

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. These fault classes constitute respectively 42% and 25% of the faults revealed by the considered checkers. The analysis of their potentially provoked errors allows us to develop some assertions detailed in Sect. 4.2.

3.2 Injected Software Faults

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 meant to facilitate the analysis. Each functional component is composed of elementary functions. It is worth distinguishing the elementary functions that are exported to the API (kernel calls) from those that are not (internal functions). We generate a call graph for each kernel call to identify the elementary functions called by the considered kernel call. For each kernel call, we define depth levels. As an example, Fig. 2 describes the call graph for the kernel call sched_setscheduler, which controls the per process scheduling policy. It

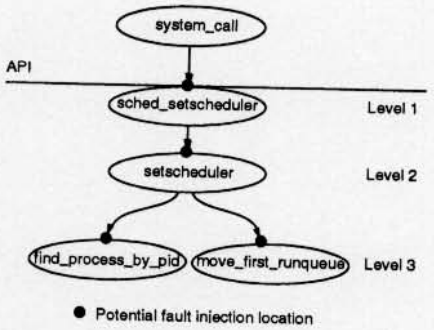

Fig. 2 Sched_setscheduler Call graph.

has three depth levels. The "system_call" node represents the kernel call entry point. It is present in all call graphs.

The goal of this study is to analyze the degree of similarity of the erroneous behaviors reported for the kernel as a consequence of fault injection at the first level (API) and in lower levels. We define a fault with respect to the i) fault location, ii) fault type and iii) trigger condition. In this paper, we consider two external fault models and two internal fault models.

External faults mimic faults from the application level and they test the robustness of the kernel. Both bit-flips and invalid parameters are injected in the parameters of the targeted kernel call. The bit-flip is a representative fault model for transient hardware faults. The goal of the proposed comparison is to determine to what extent it is equivalent to the invalid parameters fault model, which is a more focused fault model for software faults. We compare error sets caused by bit-flips, as in [16], into system call parameters with those caused by invalid parameters, as in [17]. Based on the work presented in [17], we defined eight classes of data types. For each data type class, we associate a set of invalid values.

Internal faults consist in injecting bit-flips in the values (data) or in the addresses of the parameters of the underlying kernel functions. They 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. This set of experiments targets the kernel internal functions that are not reachable via the API.

The trigger condition is the event that leads to the injection of the fault. The considered trigger condition that we have chosen is the execution of the targeted system call. The fault is thus injected depending on the selected level (external or internal).

4. Observation Levels

The objectives of the conducted experiments are to:

- Study the degree of equivalence between the two fault injection techniques at the API level,
- Compare the effects of API injected faults with those induced by 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 fault models is achieved through the comparison of their respective effects. The highest observation level (measured by *do*) consists in the quantification of the observed failure modes. If the observed failure modes, after the injection of 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 have considered to enhance the observability capabilities consists in taking into account the error detection mechanisms built into the kernel. The second refinement to further enhance the observability consists in implementing extra internal assertions within the kernel.

4.1 Experiment Outcomes

We distinguish two main types of outcome as the result of a fault injection experiment: reported and non-reported failures. The reported failures are:

- A hardware exception raised when running in kernel mode. Either the running process is killed or the kernel enters a "panic" mode,
- An application termination. If an exception is raised in user mode, the kernel terminates the process that caused the exception,
- A returned error code. 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 (termed "hindering" in [17]).

A non-reported failure is diagnosed in case of:

- A kernel hang. It can be caused by an infinite loop within the kernel or when it is waiting for an event that never occurs while interrupts are disabled,
- An application hang. 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. If the application accepts signals, we can force it to quit, otherwise we need to reboot the kernel.

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

4.2 Internal Assertions

This section presents the additional observation mechanisms that permit finer tracing of the errors provoked by the injected faults. We define a kernel control flow

as the sequence of functions called by 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 check-points by means of assertions located at the level of internal functions. A check-point 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 that do not impact the execution kernel path. The first type is based on the analysis of propagation paths of some real faults presented in Sect. 3.1. The second type consists in monitoring global kernel variables indicating the global kernel state.

As shown in Sect. 3.1, BLOCK and NULL fault classes are the most frequent software faults that have been identified in the Linux kernel. They are mostly located in the Linux device drivers. For Linux developers, while some errors can cause the kernel to enter an endless loop and thus lead it to hang, most errors manifest themselves either as null pointer dereferences or by the use of other incorrect pointer values. As a consequence, we concentrate on BLOCK and NULL classes of faults by analyzing 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 that are considered as blocking functions. We implemented assertions at 8 of these function entry points to check whether interrupts are disabled or not.

The second type of assertion gives us an internal view of the global impact of the injected faults. We select indicators for each functional component. For example, the memory pressure 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 allows us to measure this value in order to post-analyze the evolution of memory pressure during an experience.

5. Results

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. The experiment outcomes such as the kernel or the application hang are stored on the host machine. This test bed is described in detail in [10].

We carried out 5525 experiments which targeted the scheduling, memory, synchronization and communication components. The workloads used were selected to activate the kernel's functional components similarly to normal application behavior. Bit-flips in internal function parameters target 183 internal functions. These internal functions belong to the call graphs of the experiment's target kernel calls. Table 1 summarizes the distribution of experiments per functional component. The number of experiments when injecting invalid values is less than when injecting bit-flips. In fact, an average of 4 invalid values are associated to each kernel call parameter for the former injection technique, while 32 bit-flips are issued for each kernel call parameter for the latter. This means that experiments targeting invalid parameters last 8 times less time. Although, they require more time for preparation.

5.1 Analysis of the Failure Modes

Figure 3 presents the observed outcomes. We carried out three kinds of analyses. The first one is meant to compare the provoked errors of the external fault models. We then compare the errors provoked by all the injected fault models. Finally, we present details about the influence of the target kernel functional component.

The two external fault models provoke approximately the same failure modes in terms of nature and

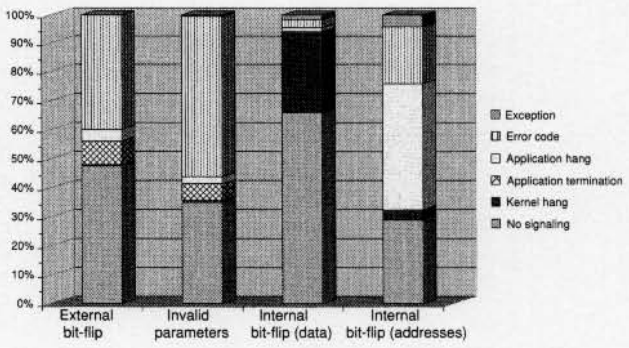

Fig. 3 Failure mode distributions.

Table 1 Experiment distribution.

	Bit-flip in API parameters	Invalid parameters	internal Bit-flip (data)	internal Bit-flip (address)
scheduling	1466	185	97	80
memory	1024	168	567	230
synchronization	448	21	107	91
shared memory	256	39	351	395
total	3194	413	1122	796

quantity. The dominant failure mode is returning error codes (44% and 55% respectively for bit-flip in API parameters and invalid parameters). 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 Sect. 5.2.1. API invalid parameters provoke fewer "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 the difference in the generated failure modes between the injections at the kernel API level and in its internal functions. The proportion of error code outcomes when injecting inside the kernel is low (3% and 18%).

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 Fig. 2. For example sched_setscheduler calls the setscheduler function, which fulfills the required service. One might expect that injections at the second depth level of this kind of kernel calls 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, which sets kernel timers, 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 3 shows that 28% of the injected faults in internal function parameter values (data) lead to a kernel hang, which is significantly higher than the 0.7% observed when injecting external faults. Also, 43% of the faults injected in internal function parameter addresses lead to application hang, which is much higher than for external faults (4% and 2%).

5.2 Analysis of the Returned Error Codes

Based on the returned error codes, we carried out two analyses: analysis of the nature of error codes and analysis of their propagation.

5.2.1 Error Codes Provoked

Figure 4 shows that the proportion of error code outcomes is greater when injecting invalid parameters than when injecting bit-flips. Figures 4-a and 4-b refine these

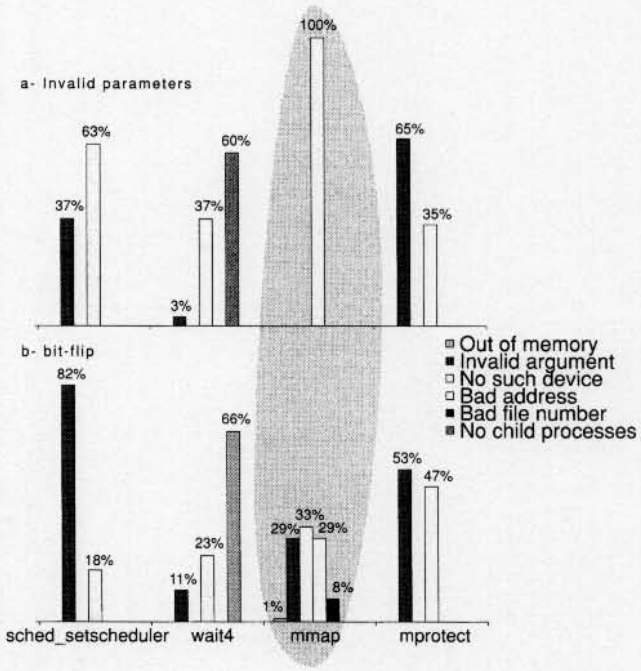

Fig. 4 Returned error code analysis.

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. However, we notice a slight advantage for the bit-flip injection technique that provides a wider variety of error codes than the invalid parameters. Also, the distribution of error codes is not always equivalent, except for certain cases such as wait4, which suspends execution of the current process until a children process has exited.

The mmap kernel call, used to map process memory, displays a singular behavior. As illustrated in Fig. 4, 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").

5.2.2 Error Propagation

Our experiments have targeted four functionnal components (scheduler, memory, synchronization and communication). Figure 5 presents the rates of errors propagated between the five functionnal components. The analysis of error propagation paths whithin the kernel is an interesting dependability benchmark measure, since it determines the error confinement degree associated to each functional component. The first percentage to each rate is associated with bit-flip experiments, and the second percentage is associated to invalid parameters. Injection in the memory component provokes propagation to the scheduler and the file system functional components. The injection in the scheduler and the sychronization provokes propagation within the

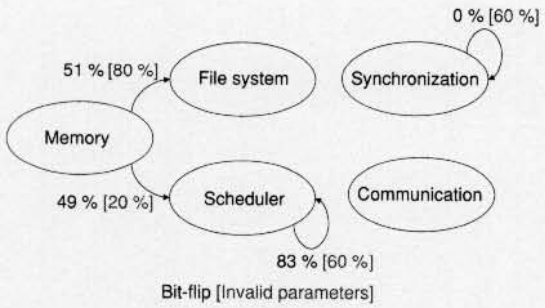

Fig. 5 Error propagation.

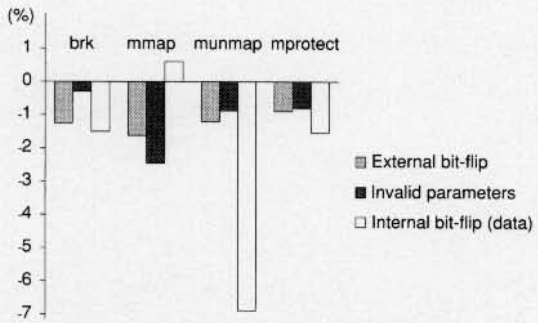

Fig. 6 Brk failure rate variation.

functional component. The injections in the communication component do not provoke propagation.

5.3 Assertion Analysis

We detail hereafter the observations made thanks to the implemented assertions. We recall that these assertions are derived from the analysis of the source code. We are considering successively the two assertions that led 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 does not have to be changed in 64% of the cases (brk doesn't carry out a specific treatment). Figure 6 illustrates the cases where we noticed a deviation from this normal invocation. Injected faults in the internal function parameter values, 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 injecting 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

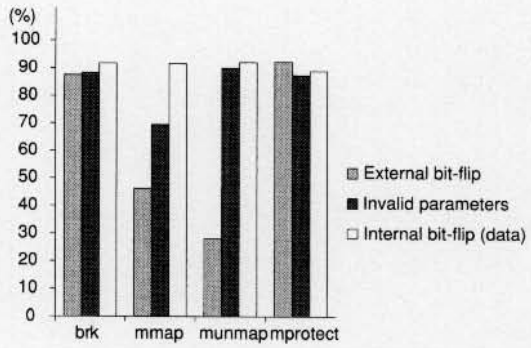

Fig. 7 Memory pressure variation.

that the kernel is trying to satisfy and thus the current memory load. The greater this value, the more the memory resource becomes a critical issue. Figure 7 presents the percentages of experiments where the memory pressure increases by more than 50% after a fault injection in the memory kernel calls. 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 parameter values on the "memory pressure" is constant. For all the memory kernel calls, about 90% of the experiments lead to a memory pressure increase. 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.

5.4 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 Fig. 3, only 0.3% and 0.7% of the external faults, lead to "Kernel hang", while 28% and 3% 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 fault leads to an "Exception". Only 0.01% and 0.5% of the external faults lead to an "Exception", while 1.6% and 4.3% of internal faults provoke an "Exception".

In addition to failure modes, the implemented assertions give a complementary observation viewpoint. Their goal is to observe the possible correlation between

real errors (caused by real faults) and errors provoked by the four 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 any of the four considered 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 ongoing to better assess the representativeness of injected faults with respect to real faults affecting device drivers.

6. Summary and Conclusion

This work compares the impact of four types of SWIFI techniques on Linux. 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 other two apply bit-flips in the parameters values (data) and the addresses of the internal functions of the kernel.

In addition to the observed outcomes, specific assertions were implemented to provide finer grained reporting. Some of these assertions were deduced from the analysis of the effects caused by real faults.

API-level fault injection is a 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, its application is more time consuming, as it needs 32 injections per parameter for a 32-bit kernel and simple data types. An invalid parameter campaign is 8 times faster than a bit-flip one.

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. Also, the proportion of experiments that lead to an increase of the memory pressure is greater when injecting invalid parameters.

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, for example for all POSIX compliant systems.

Compared to the effects induced by external faults, flipping bits in internal function parameters provoked distinct erroneous behaviors. Indeed, the proportion of error codes observed was lower than for the external fault injection 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 such 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 each functional component.

Acknowledgement

This work has largely benefited from fruitful discussions with Jean-Claude Laprie from LAAS-CNRS. Also, we would like to thank Moslem Belkhiria and Benjamin Lussier who contributed in the experiment setup.

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 Trans. Softw. Eng., vol.16, no.2, pp.166–182, Feb. 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," Proc. 26th Int. Symp. on Fault-Tolerant Computing (FTCS-26), pp.353–357, Sendai, Japan, 1996.

[3] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," IEEE Trans. Comput., vol.42, no.8, pp.913–923, Aug. 1993.

[4] A. Arazo and Y. Crouzet, "Formal guides for experimentally verifying complex software-implemented fault tolerance mechanisms," Proc. 7th Int. Conf. on Engineering of Complex Computer Systems (ICECCS2001), pp.69–79, Sweden, 2001.

[5] J.V. Carreira, D. Costa, and J.G. Silva, "Fault injection

spot-checks computer system dependability," IEEE Spectr., vol.36, pp.50–55, Aug. 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] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G.H. Leber, "Comparison of physical and software-implemented fault injection techniques," IEEE Trans. Comput., vol.52, no.9, pp.1115–1133, Sept. 2003.
[8] H. Madeira, K. Kanoun, J. Arlat, Y. Crouzet, A. Johanson, and R. Lindstrom, "Preliminary dependability benchmark framework," DBench Project IST 2000-25425 Deliverable. CF2, Available at http://www.laas.fr/dbench/delivrables.html, 2001.
[9] J.C. Laprie, "Dependable computing: Concepts, limits, chalenges," Proc. 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25), pp.42–54, Pasadena, CA, 1995.
[10] T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun, "Experimental analysis of the errors induced into Linux by three fault injection techniques," Proc. Int. Conf. on Dependable Systems and Networks (DSN-2002), Washington, DC, 2002.
[11] 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 Trans. Nucl. Sci., vol.47, no.6, pp.2231–2236, Dec. 2000.
[12] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults-based on field data," 26th Int. Symp. on Fault Tolerant Computing, Sendai, Japan, 1996.
[13] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," Proc. Int. Conf. on Dependable Systems and Networks (DSN-2000), pp.417–426, NY, 2000.
[14] I. Lee and R.K. Iyer, "Software dependability in the Tandem GUARDIAN system," IEEE Trans. Softw. Eng., vol.21, no.5, pp.455–467, 1995.
[15] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific programmer written compiler extensions," Proc. 4th Symp. on Operating Systems Design and Implementation (OSDI-2000), San Diego, CA, 2000.
[16] M. Rodriguez, F. Salles, J.-C. Fabre, and J. Arlat, "MAFALDA: Microkernel assessment by fault injection and design aid," Proc. 3rd European Dependable Computing Conf. (EDCC-3), pp.143–160, Prague, Czech Republic, 1999.
[17] P.J. Koopman, J. Sung, C. Dingman, D.P. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," Proc. 16th Int. Symp. on Reliable Distributed Systems (SRDS-16), pp.72–79, Durham, NC, 1997.
[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 Trans. Softw. Eng., vol.18, no.11, pp.943–956, Nov. 1992.

Tahar Jarboui received the engineer degree from l'Ecole Nationale d'Ingénieurs de Tunis, Tunisia. In 2003, he received his doctorate from the Institut National Polytechnique of Toulouse, France, that he prepared in the Dependable Computing and Fault-Tolerance group at LAAS-CNRS. He currently holds a postdoctoral research position there. His research interests focus on operating system dependability benchmarking.

Jean Arlat is Directeur de Recherche at CNRS, the French National Organization of Scientific Research and leads the research group on Dependable Computing and Fault Tolerance at LAAS-CNRS. His research interests focus on the dependability of hardware-and-software fault-tolerant systems and the dependability evaluation of off-the-shelf software operating systems, including both analytical modeling and experimental fault injection approaches.

Yves Crouzet is currently Chargé de Recherche at CNRS and a member of the Dependable Computing and Fault-Tolerance group at LAAS-CNRS. He initially worked on the design and realization of self-checking VLSI circuits. His main research interests now concern the experimental validation of dependable systems by fault-injection and the experimental validation of software testing methods by mutation analysis.

Karama Kanoun is Directeur de Recherche at CNRS and a member of the Dependable Computing and Fault-Tolerance group at LAAS-CNRS. Her research interests include modeling and evaluation of computer system dependability considering hardware as well as software. Her recent research focus on Dependability Benchmarking: she leads the IST project DBench and chairs the IFIP WG 10.4 SIG on this topic.

Thomas Marteau received the engineer degree from l'Ecole Supérieure d'Ingénieurs en Electrotechnique et Electronique, Paris, France. He has worked in the Dependable Computing and Fault-Tolerance group at LAAS-CNRS to prepare this degree. He is now with PSA-Peugeot-Citroën.