Do Injected Faults Cause Real Failures? — A Case Study of Linux —

Nobuo Kikuchi, Takeshi Yoshimura, Ryo Sakuma, Kenji Kono
Keio University

***Abstract*—Software fault injection (SFI) has been used to intentionally cause "failures" in software components and assess their impacts on the entire software system. A key property that SFI should satisfy is the *representativeness* of injected failures; the failures caused by SFI should be as close as possible to failures in the wild. If injected failures do not represent realistic failures, the measured resilience or tolerance against failures of the investigated system is not trustworthy. To the best of the authors' knowledge, the representativeness of "faults" has been investigated. However, it is an open problem whether the failures caused by injected faults represent realistic failures. In this paper, we report the preliminary results of the investigation on the representativeness of injected failures. To compare injected failures with real failures, we have collected 43,742 real crash logs of Linux from the RedHat repository, and conducted a fault injection campaign on Linux, using SAFE, a state-of-the-art injector of software faults. In the fault injection campaign, 50,000 faults are injected to the Linux file system and 71,470 runs of a workload are executed. The crash logs generated by SFI are compared with the real RedHat logs with respect to crash causes, crashed system calls, and crashed modules. Our preliminary results suggest that failures caused by injected faults do not represent real failures, probably because injected faults are not representative enough or because the selected workload is not realistic.**

I. Introduction

Software fault injection (SFI) has been widely used to intentionally inject faults to software components and assess the impact of software failures caused by those faults on the entire software system [1]. A key property that SFI should satisfy is the *representativeness* of injected failures; the failures caused by SFI should be as close as possible to failures in the wild. If injected failures do not represent realistic failures, the measured resilience or tolerance against failures of the investigated system is not trustworthy. Suppose that a fault injection campaign concludes 90% of injected failures are gracefully handled by a target fault-tolerant mechanism. If the injected failures are not representative of realistic failures and compose only 10% of the real ones, the target fault-tolerant mechanism can handle only 0.9% of real failures in practice. To develop better ways of injecting faults, it is mandatory to investigate to what extent the state-of-the-art fault injectors can reproduce representative failures.

To the best of the authors' knowledge, the representativeness of "faults" has been investigated [11]. If injected faults are not representative of real faults, the resulting failures are not representative of real failures. But even if injected faults are representative of real faults, it is not clear that failures caused by those faults are also representative of real failures. For example, a single fault may produce various failures depending on how the fault is activated. In the previous work [11], if injected faults can not be detected through test cases adopted by developers, they are considered representative because they tend to be residual faults in deployed software systems. In addition to this property of injected faults, "failures" caused by injected faults should be as close as possible to realistic failures.

Our concern in this paper is that failures caused by the state-of-the-art fault injector reproduce realistic failures. Aside from the fault injector, it is important to choose a realistic workload to reproduce realistic failures. In this paper, we assume that benchmark programs in wide use are realistic at the reasonable level because designing a benchmark program that generates realistic workloads is another issue. So, even if our experimental results suggest that injected failures are not representative, it does not necessarily mean injected faults are not representative; there is the possibility that the workloads generated by benchmarks are not realistic. Our claim is that the "failure" representativeness must be taken into consideration when a fault campaign is carried out to assess fault resilience or tolerance of software systems.

The target software in this paper is the Linux operating system (OS). An operating system is the most fundamental software. An OS crash causes all the applications on top of it to fail. If mission-critical services are running, millions of dollars can be lost [3]. Unfortunately, Linux is far from bug-free [4]. To tolerate residual faults in operating systems, many fault tolerance mechanisms have been proposed for the Linux kernel [5], [6], [7], [8], [9]. The effectiveness of those mechanisms are often evaluated through SFI, believing that SFI injects realistic faults and causes representative failures. If failures caused by injected faults are not representative, all the evaluation results are not trustworthy.

Research efforts in the past about fault representativeness have improved the representativeness of faults [10], [11]. Representative patterns of faults are extracted by investigating bug-fix logs of major software systems [10] and fault injectors inject faults based on the extracted fault patterns. However, fault patterns do not dictate where a specific fault should be injected. [11] investigates the representativeness of injected faults. The key observation behind [11] is that residual faults do not manifest in test cases because faults that can be detected in test cases must have been fixed by developers. Unlike our work presented in this paper, these two research efforts focus

978-1-4799-7377-4/14 $31.00 © 2014 IEEE
DOI 10.1109/ISSREW.2014.104

on “fault” representativeness.

Failure representativeness is a key property SFI should satisfy because fault tolerance mechanisms deal with “failures” caused by faults. For example, ShadowDriver [5] tolerates failures caused by faults inside OS device drivers. Since buggy device drivers may crash the entire kernel if the faults inside them are activated. ShadowDriver isolates buggy device drivers from the core kernel and tolerates failures caused by buggy drivers. Therefore, whether ShadowDriver can treat a crash or not depends on a failure state of the device driver.

This paper investigates the representativeness of failures caused by injected faults. Failure states caused by SFI are compared with real failure states in this paper and how similar these two states are to each other is investigated. For real failure reports, we use 43,472 Linux crash reports collected by RedHat. To collect failures caused by SFI, an SFI campaign environment has been built. SAFE [12] is used as a fault injector because it is one of the state-of-art software fault injectors. SAFE is the latest fault injector and injects the most accurate faults at this time.

In the SFI campaign, 54,213 faults are injected to Linux file systems and the benchmark program is executed for 71,470 times. The target Linux file system are ext2, ext3, ext4, fat and xfs. UnixBench is employed for the benchmark program. In the SFI campaign, 3,626 crash reports are collected and compared with 311 crash reports in the RedHat crash reports that are related to file systems. The failure reports are analyzed from the three respects (crash cause, crashing system call, crashing module) using chi-squared test. We observed that failure states caused by SFI are different from real failure states. Under chi-squared test, p-value of the three comparisons is under 0.01. These test results show that the SFI failures are independent of the real failures; SFI do not inject realistic faults.

In section 2, we discuss the related work. In section 3, we describe the methodological approach adopted in this paper. Section 4 discusses the results from the SFI campaign. Section 5 concludes the paper.

II. Related Work

The relationship between testing and fault injection has been explored in the past. Past work focused on fault injection for generating test cases (in the hypothesis that, if a test case is effective against mutants i.e., injected faults, the test will be able to detect real faults), i.e., *mutation testing* and comparing testing strategies, i.e., *mutation analysis*. These techniques do not aim at fault forecasting or fault removal in fault tolerant systems. So injected faults and caused failures are not necessarily representative.

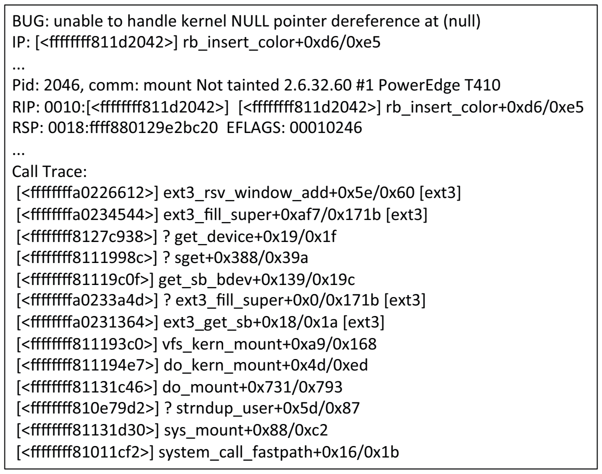
There are some examples of research of works in the literature in which the failure representativeness is crucial. The studies in [5] and [6] proposed mechanisms for improving dependability of OS i.e., ShadowDriver, Otherworld. ShadowDriver is a mechanism for device driver as described in Section 1. Otherworld is a mechanism that microreboots the operating system kernel when a critical error is encountered in the kernel. These two studies evaluated how much these mechanisms can recover OS from crashes, using the SFI technique. Failure representativeness is needed to obtain reasonable estimation of the effectiveness because inappropriate SFI can lead to wrong estimation of the effectiveness. If the failures that are not realistic happen so often during the campaign, the resulting evalutaion results is better than the actual benefit that the proposed mechanism can offer; the mechanism is overestimated. The SFI technique used in these two works is originally developed for evaluating the reliability of the Rio File Cache [13], in which the fault model is based on an old study [14]. Since the fault model is based on the old study in 1992, the target OSes investigated there are VAX/VMS and Tandem GUARDIAN which have less features compared with modern OSes like Linux and assume more simple hardware devices (no multi-core CPUs and so on).

Software fault representativeness is addressed in [10], [11], [12]. The study in [10] explores 668 bug-fix patches to investigate the fault patterns in major software systems. The patches they investigate include bug fixes of Linux kernel, vim, bash and other widely used software systems. According to the paper, a new fault category is identified and a new binary-level SFI tools called G-SWFIT is proposed. In [11], fault representativeness of G-SWFIT is evaluated. Residual faults are not detected even if many test cases are executed during the software test. The study assumes the faults that are activated in a small set of test cases are not elusive. If less than 50% of test cases detect the fault, it is an elusive fault; otherwise, it is not elusive. The study in [12] evaluates the binary-level SFI technique, G-SWFIT, from the aspect that G-SWFIT can accurately inject faults to the code locations defined in [10] with the comparion to the source-code-level SFI technique, SAFE, that can inject faults more accurately than G-SWFIT because of the source-level information. And they show that G-SWFIT does not inject faults accurately than SAFE. G-SWFIT is useful because it can deal with executable code without source code but does not show good accuracy.

In the past there is no detailed study on *failure representativeness*. This is the first attempt to investigate failure representativeness by SFI.

III. Research methodology

This work aims to figure out SFI technique can achieve failure representativeness, using the Linux operating system as a case study. To approach the question, we compare real crash logs and those obtained through SFI. If SFI achieves good representativeness of failures, these two types of crash logs have the characteristics similar to each other. Linux kernel crash log is called an oops message. An example of the oops massages is shown in Fig [1]. An oops message has the following information. At the first line, the cause of the crash is described. In this case, the kernel cannot handle NULL pointer dereference in the kernel space. In addition, an oops message has the call trace information. RIP line describes a function in which the kernel crashed and other trace information is described under Call Trace line. The



```
BUG: unable to handle kernel NULL pointer dereference at (null)
IP: [<ffffffff811d2042>] rb_insert_color+0xd6/0xe5
...
Pid: 2046, comm: mount Not tainted 2.6.32.60 #1 PowerEdge T410
RIP: 0010:[<ffffffff811d2042>]  [<ffffffff811d2042>] rb_insert_color+0xd6/0xe5
RSP: 0018:ffff880129e2bc20  EFLAGS: 00010246
...
Call Trace:
 [<ffffffffa0226612>] ext3_rsv_window_add+0x5e/0x60 [ext3]
 [<ffffffffa0234544>] ext3_fill_super+0xaf7/0x171b [ext3]
 [<ffffffff8127c938>] ? get_device+0x19/0x1f
 [<ffffffff8111998c>] ? sget+0x388/0x39a
 [<ffffffff81119c0f>] get_sb_bdev+0x139/0x19c
 [<ffffffffa0233a4d>] ? ext3_fill_super+0x0/0x171b [ext3]
 [<ffffffffa0231364>] ext3_get_sb+0x18/0x1a [ext3]
 [<ffffffff811193c0>] vfs_kern_mount+0xa9/0x168
 [<ffffffff811194e7>] do_kern_mount+0x4d/0xed
 [<ffffffff81131c46>] do_mount+0x731/0x793
 [<ffffffff810e79d2>] ? strndup_user+0x5d/0x87
 [<ffffffff81131d30>] sys_mount+0x88/0xc2
 [<ffffffff81011cf2>] system_call_fastpath+0x16/0x1b
```

Fig. 1. oops message sample

kernel version information is also described. We use these pieces of information to compare crash logs. To obtain crash logs by SFI, we construct an SFI campaign environment. We use the state-of-the-art SFI technique because any other SFI techniques neither achieve the representativeness if it does not achieve it.

IV. Evaluation of Failure Representativeness

In this section, we first provide the details about the case study and the experimental setup. Then, we discuss the results we obtained.

A. *Case study and experimental setup*

Linux is one of the most widely used OSes and many fault tolerant mechanisms have been proposed for the Linux kernel. We use the Linux long term supported (LTS) version i.e., Linux 2.6.32 because the LTS version is believed stable and thus, widely used in the wild, and many real crash logs are available. And we focus on Linux file system since it is one of the buggies modules in the Linux kernel [4]. We compare real crash logs and crash logs by SFI. For real crash logs, we use 43,742 crash logs collected by RedHat for one year from September 2012. Then we select 311 real crash logs that are related with Linux 2.6.32 file systems. If a call trace includes a function name that resides in the file system modules, we regard the log is related to faults in the file system. The Linux version is obvious from the description in the oops message.

We use SAFE as the state-of-the-art fault injector. SAFE is a source-level fault injector. Fig. 2 shows the behavioral flow of SAFE. First, SAFE analyzes the target source code and finds the fault injection points based on the fault operation library information. This fault operation information is originally defined for G-SWFIT in [10] and SAFE is developed in [12] as a fault injector that can inject faults as in G-SWFIT at the source-code level. This information includes where to inject fault and how to change the source code. All the operations are shown in Table I. SAFE injects 13 types of faults based on

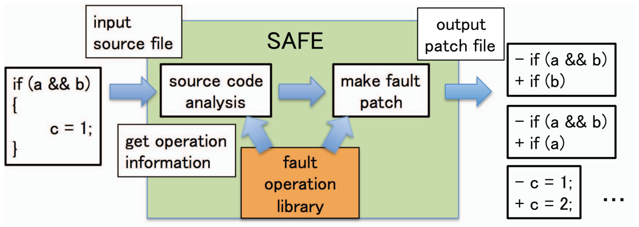
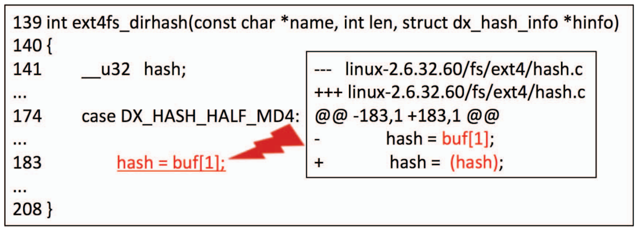


Fig. 2. fault injector SAFE behavior flow

TABLE I
FAULT INJECTION OPERATION OF SAFE

Operator	Fault type addressed
OMFC	MFC - Missing Function call
OMVIV	MVIV - Missing variable initialization using a value
OMVAV	MVAV - Missing variable assignment using a value
OMVAE	MVAE - Missing variable assignment using an expression
OMIA	MIA - Missing if construct around statements
OMIFS	MIFS - Missing if construct plus statements
OMIEB	MIEB - Missing if construct plus statements plus else before statements
OMLAC	MLAC - Missing ”AND EXPR” in expression used as branch condition
OMLOC	MLOC - Missing ”OR EXPR” in expression used as branch condition
OMLPA	MLPA - Missing small and localized part of the algorithm
OWVAV	WVAV - Wrong value assigned to variable
OWPFV	WPFV - Wrong variable used in parameter of function call
OWAEP	WAEP - Wrong arithmetic expression in parameter of a function call



```
int ext4fs_dirhash(const char *name, int len, struct dx_hash_info *hinfo)
{
        __u32   hash;
...
        case DX_HASH_HALF_MD4:
...
                hash = buf[1];
...
}
```

```
--- linux-2.6.32.60/fs/ext4/hash.c
+++ linux-2.6.32.60/fs/ext4/hash.c
@@ -183,1 +183,1 @@
-           hash = buf[1];
+           hash =  (hash);
```

Fig. 3. fault patch example : OMVAE

this operational information. After the source code analysis, SAFE makes patches to inject faults to the target source code. The faults are injected by applying the patches to the source files before the target sources are compiled. An example of the fault injection patch generated by SAFE is shown in Fig. 3. In Fig. 3, the value of expression `buf[1]` is assigned to variable `hash`. OMVAE is a fault injection operation that emulates the missing variable assignment. Here, the variable assignment for `hash` is omitted by changing code as `hash = (hash)`. SAFE injects faults by making patches that perform such operations.

We target 5 major file systems in Linux 2.6.32.60; ext2, ext3, ext4, fat and xfs. Many patches for these file systems are generated but we observe some of

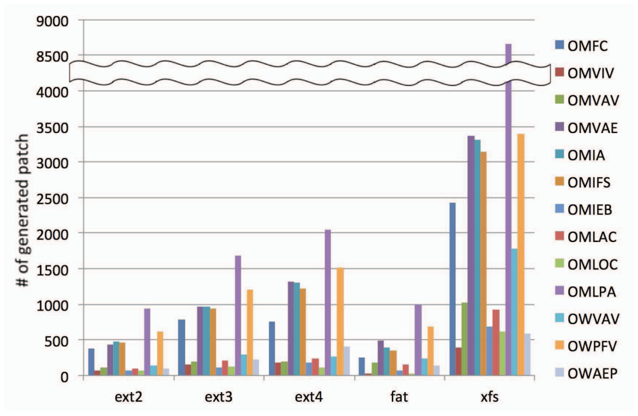


Fig. 4. number of patches generated by SAFE at each filesystem

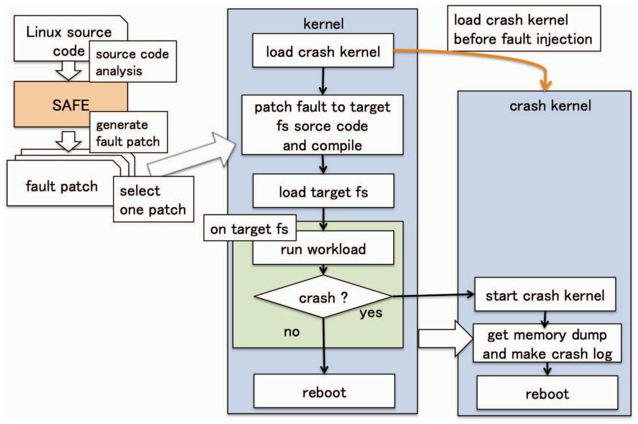


Fig. 5. SFI campaign flow

those patches are not suitable. The patches modify C include files such as `include/linux/mm.h` or `arch/x86/include/asm/processor.h` that affects source files that are not included in `fs` directories. Such patches do not inject faults related to file systems and removed them. So we used last 55,878 patches related with filesystem. 55,878 patches and show detail in Fig 4. For each file system, the distribution of each patch tends to be the same and the number of patches seems to be directly proportional to the number of lines of code in the file system.

For collecting crash logs by SFI, we should solve two problems. In this study, the kernel may not produce correct crash logs because of our injected faults to the kernel. So we use the kdump tool to collect crash logs. Kdump is a tool to collect logs aside from the crashing kernel. It prepares another kernel called a crash kernel that collects crash logs before the kernel crashes. Since faults are not injected to the crash kernel, it can collect crash logs correctly. Another problem is that we need many crash logs to analyze crash logs statistically. An automatic crash log collecting environment has been built. Then we construct a SFI campaign environment. The details of the built environment is shown in Fig. 5. After booting the

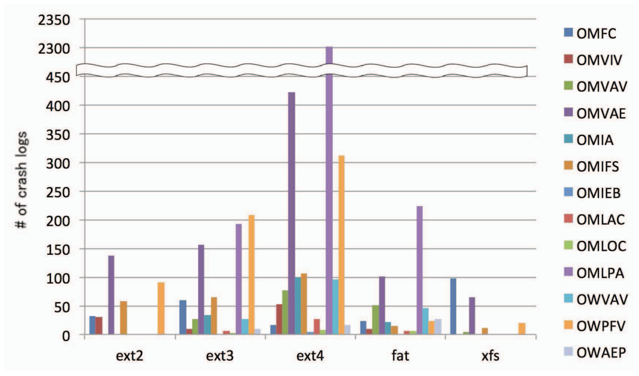


Fig. 6. The number of crashes obtained for each type of fault pattern

TABLE II
SFI CAMPAIGN SUMMARY

filesystem	# of patches	# of experiment	# of crash logs
ext2	3939	10761	382
ext3	7855	29300	782
ext4	9736	16611	3532
fat	4005	14231	550
xfs	30323	567	224

machine for the SFI campaign, faults are injected to target file systems by applying a patch file one by one. Then, UnixBench benchmark program is executed on the target file system. If the target file system crashes, the crash kernel collects the crash log. Finally, the machine is rebooted to continue the campaign. We perform the fault injection for 71,470 times since August 2013 to July 2014 on 10 physical machines. As a result, 3,626 crash logs are collected. The details are shown in Fig. 6. At all file systems, the number of the generated patches is not proportional to the number of the obtained crash logs. The summary of the SFI campaign is shown in Table II. All the patches for ext2, ext3, ext4 and fat are used at least once. Our SFI campaign system does not work well for the xfs environment. So we only applied 567 patches and obtained 224 crash logs.

B. *Results*

We compare the crash logs from the following three aspects: crash cause, system call, and the module that is called at the top of the call trace.

First, we discriminate crash causes. The crash causes discussed in this study are as follows:

- *Unable to handle kernel paging request at NULL pointer dereference*: this message is emitted when the kernel cannot handle the deferences of pointers to the addresses in the virtual page 0. This crash cause is referred to as NULL in this paper.
- *Unable to handle kernel paging request at pointer dereference*: this message is emitted when the kernel cannot handle pointer dereferences on the addresses other than

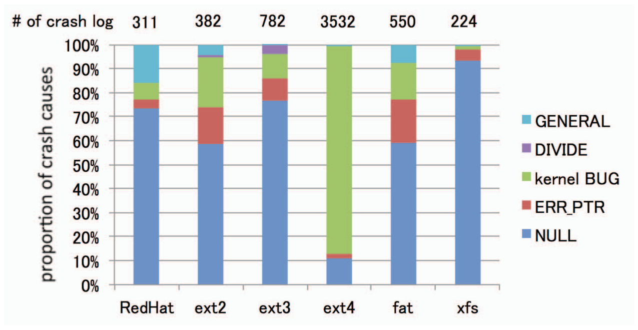


Fig. 7. proportions of crash causes per real and SFI environment

in virtual page 0. This fault cause is referred to as ERR_PTR.

- *General protection fault*: this message is emitted when a general protection fault of Intel x86 processors occurs in the kernel. This is referred to as GENERAL.
- *Divide by zero* this message is emitted if the kernel divides a value by zero. Referred to as DIVIDE.
- *kernel BUG*: this message is emitted if the kernel execute the BUG_ON() function.

The results are shown in Fig. 7. In Fig. 7, the crash causes in the RedHat crash logs are shown as RedHat. Other file system names indicate the results from the SFI campaign that targets each file system shown in the table. We observe there are some differences in the proportion of the crash causes between RedHat results and the others. For example, there is no DIVIDE in the real RedHat environment but there are some DIVIDE failures in the ext3 SFI environment. ERR_PTR proportion in the SFI environment except for xfs and ext4 is bigger than the real RedHat environment and kernel_BUG accounts for over 80% in the ext4 environment but only a few accounts in the real RedHat environment. We apply the two-sample chi-squared test to each result (the real result and one of the SFI campaign results). And we observe all the p-values are under 0.01. In the chi-squared test, the null hypothesis is that the two sampling distributions are dependent on each other. In this comparison, since the p-value is under 0.01, the null hypothesis is rejected. From the perspective of the crash cause, the SFI campaign shows the statistical tendency that differs from the real RedHat crash logs. This implies that SFI cannot achieve the failure representativeness.

Second, we compare the system calls that lead to kernel crashes. The results are shown in Fig. 8. In Figs. 8 and 9, the number of crash logs differs from that in 7 because some oops messages are not perfect. In 8, we observe there are different tendencies about system call proportions. For example, `sys_mkdir` accounts for little proportion in the RedHat environment but it accounts for some proportion in the SFI environments. And `sys_umount` accounts for less proportion in the RedHat environment but it accounts for some proportion in ext2 and ext3 and accounts for most of the proportion in ext4. Two-sample chi-squared test is applied to the results of this comparison, and all the p-values are under 0.01. From the perspective of failing system calls, the SFI technique does not achieve the failure representativeness.

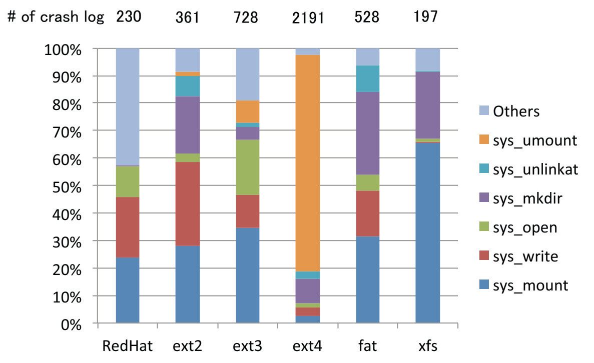


Fig. 8. Proportions of the called system calls for RedHat and SFI environments

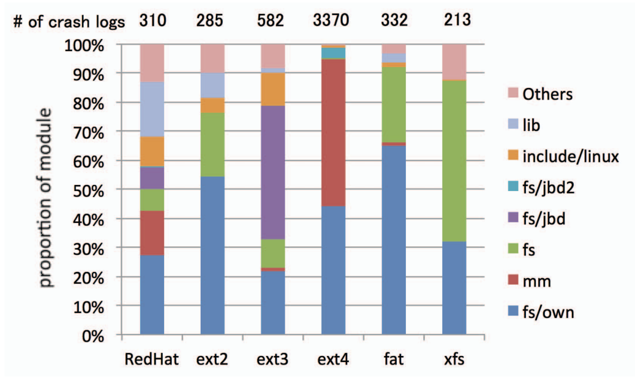


Fig. 9. Proportions of the modules that defines functions on the top of stack trace

Finally, we compare the modules that invoke the function on top of the call trace. Since the number of functions that are on top of the call traces, we compare the distribution of modules that call the functions on top of the trace. To determinate the modules that invoke the final functions, we regard a module in which the function is defined is the module that invokes the function. The results are shown in Fig. 9. In Fig. 9, `fs/own` indicates the target file system in the environment (In the SFI environment that targets ext4, fs/own means fs/ext4). In the RedHat crash logs, `fs/own` indicates all the file systems. There are notable differences in the RedHat and SFI environments. For example, `lib` accounts for about 20% in the RedHat environment but at most 9% in the SFI environments. And `mm` accounts for about 15% in the RedHat environment but at most 5% in the SFI environments except for ext4. We apply the two-sample chi-squared test to this comparison, and observe all p-values are under 0.01. Again, SFI technique does not achieve the failure representativeness regarding the top functions of the call traces.

V. Conclusion

This paper investigates the representativeness of failures that are caused by the state-of-the-art software fault injections. In this paper, the Linux operating system has been empoyed as a case example of the study because the Linux is a "real" software system that is widely used in the wild. To evaluate the failure representativeness, the real crash logs collected by RedHat for almost one year and the crash logs generated by SAFE, one of the state-of-the-art software fault injectors are statistically compared. To compare the similarities of the crash logs, three different aspects of the logs are compared; crash causes, failing system calls, and modules in which the kernel crashes (i.e.; the top function of the call trace). The two-sample chi-squared test is applied to the RedHat crash logs and the crash logs generated in the SFI environments. The results of all the tests are negative; all the p-values are under 0.01. This implies that the current fault injectors do not achieve good representativeness of failures. Since the failures caused by a specific fault depends on the workloads, we have to further investigate what happens if we change the benchmark programs that are executed in the SFI environments.

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," *Software Engineering, IEEE Transactions on*, vol. 16, no. 2, pp. 166–182, 1990.

[2] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *Software Engineering, IEEE Transactions on*, vol. 39, no. 1, pp. 80–96, 2013.

[3] J. Jann, R. S. Burugula, C. Wu, and K. El Maghraoui, "An os-hypervisor infrastructure for automated os crash diagnosis and recovery in a virtualized environment," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*. IEEE, 2012, pp. 195–202.

[4] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in linux: Ten years later," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 305–318.

[5] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 4, pp. 333–360, 2006.

[6] A. Depoutovitch and M. Stumm, "Otherworld: giving applications a chance to survive os kernel crashes," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 181–194.

[7] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, "Nooks: An architecture for reliable device drivers," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM, 2002, pp. 102–107.

[8] F. M. David, E. Chan, J. C. Carlyle, and R. H. Campbell, "Curios: Improving reliability through operating system structure." in *OSDI*, 2008, pp. 59–72.

[9] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Fine-grained fault tolerance using device checkpoints," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1. ACM, 2013, pp. 473–484.

[10] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *Software Engineering, IEEE Transactions on*, vol. 32, no. 11, pp. 849–867, 2006.

[11] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira, "Representativeness analysis of injected software faults in complex software," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. IEEE, 2010, pp. 437–446.

[12] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *Dependable Computing Conference (EDCC), 2012 Ninth European*. IEEE, 2012, pp. 162–172.

[13] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, *The Rio file cache: Surviving operating system crashes*. ACM, 1996, vol. 31, no. 9.

[14] I. Lee, D. Tang, R. K. Iyer *et al.*, *Measurement and Analysis of Operating System Fault Tolerance*. Citeseer, 1992.

[15] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[16] H. Stuart, "Hunting bugs with coccinelle," *Master's Thesis*, 2008.

[17] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer, "Failure data analysis of a lan of windows nt based computers," in *Reliable Distributed Systems, 1999. Proceedings of the 18th IEEE Symposium on*. IEEE, 1999, pp. 178–187.

[18] I. Lee and R. K. Iyer, "Software dependability in the tandem guardian system," *Software Engineering, IEEE Transactions on*, vol. 21, no. 5, pp. 455–467, 1995.

[19] M. Sullivan and R. Chillarege, "A comparison of software defects in database management systems and operating systems," in *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*. IEEE, 1992, pp. 475–484.

[20] J. Gray, "A census of tandem system availability between 1985 and 1990," *Reliability, IEEE Transactions on*, vol. 39, no. 4, pp. 409–418, 1990.