Is Linux Kernel Oops Useful or Not?

Takeshi Yoshimura
Keio University

Hiroshi Yamada
Keio University, CREST/JST

Kenji Kono
Keio University, CREST/JST

Abstract

Linux kernel oops is invoked when the kernel detects an erroneous state inside itself. It kills an offending process and allows Linux to continue its operation under a compromised reliability. We investigate how reliable Linux is after a kernel oops in this paper. To investigate the reliability after a kernel oops, we analyze the *scope* of error propagation through an experimental campaign of fault injection in Linux 2.6.38. The error propagation scope is *process-local* if an error is confined in the process context that activated it, while the scope is *kernel-global* if an error propagates to other processes' contexts or global data structures. If the scope is process-local, Linux can be reliable even after a kernel oops. Our findings are twofold. First, the error propagation scope is mostly process-local. Thus, Linux remains consistent after a kernel oops in most cases. Second, Linux stops its execution before accessing inconsistent states when kernel-global errors occur because synchronization primitives prevent the inconsistent states from being accessed by other processes.

1 Introduction

Linux kernel oops is invoked when the kernel detects an erroneous state inside itself. It prints out an oops message and kills the offending process to allow Linux to continue its operation under a compromised reliability. After the kernel oops occurs, nothing is guaranteed because no one can tell which kernel states are consistent or not. If an inconsistent state happens to be confined in the context of the offending process, Linux is expected to be reliable even after the oops because the inconsistent state can be revoked by killing the offending process. Otherwise, the kernel becomes unreliable since its operation is based on inconsistent states.

We investigate how reliable Linux can be after a kernel oops in this paper. We introduce the concept of the *scope* of error propagation to investigate the reliability after a kernel oops. The error propagation scope is *process-local* if an error is confined in the process context that activated it. The scope is *kernel-global* if an error propagates to other processes' contexts or global data structures.

The distinction between process-local and kernel-global propagation is significant. If an error is process-local, the kernel oops allows us to recover from the error because it revokes the inconsistent states by killing the faulty process. If an error is kernel-global, the recovery is hopeless because corrupted global data structures must be recovered in order to continue processing.

We conduct a series of fault injection experiments to investigate the scope of error propagation. Since our target is Linux bugs, we use a fault injection tool widely used in the OS community [13, 12, 3, 14, 9]. This fault injector focuses on the emulation of low- and high-level software faults, including errors specific to operating system kernels.

In our fault injection experiments, 6,738 faults (15 types of faults) are injected into Linux 2.6.38. The kernel oops is invoked in 134 out of 6,128 faults, including panic(). We follow an execution trace until either an oops or panic() is called in order to analyze the error propagation when an injected fault is activated. We track down how an injected error propagates in the kernel using the execution trace.

According to our experimental results, the Linux kernel oops is useful in handling kernel failures for the following reasons.

- The scope of error propagation is mostly process-local in Linux. This implies that the Linux kernel oops is effective in recovering from kernel failures in most cases. Since an error is not propagated to other process contexts, the kernel can be recovered to a consistent state simply by revoking the context of the faulty process.

- Even if an error propagates to shared kernel data structures, the non-faulty processes do not access the inconsistent data because the faulty process crashes inside a critical section with a lock acquired. This suggests that Linux shows fail-stopness even when there are kernel-global errors.

The rest of this paper is organized as follows. Section 2 describes the work related to ours. Section 3 explains the software fault injector used in the experiments. Section 4 reports our experimental results. Section 5 concludes this paper.

2 Related Work

Linux is far from bug-free. In practice, it is almost impossible to eliminate all the bugs in Linux, despite the tremendous advances made in debugging tools, testing methodologies, static analysis, and formal methods. According to the empirical study on Linux bugs [10, 1], the number of bugs per line is decreasing but the increased size of the Linux code makes the total number of bugs almost constant. Microkernel-based OSes are expected to be more reliable. Minix3 can isolate error propagation [6] and the seL4 can be formally verified to be correct [7].

Software-implemented fault injection (SWIFI) has been conducted with emphasis placed on the different aspects of fault manifestation to better understand the kernel behavior under fault manifestation. Our focus in this paper is the "scope" of error propagation (i.e., process-local or kernel-global). Previous work focuses on other aspects of error propagation than the scope of error propagation.

Gu *et al.* [5] use SWIFI to characterize Linux behaviors under error manifestation. Their analysis shows that crash latencies are within 10 cycles in most cases and also shows how an error propagates between OS subsystems. Our concern in this paper is that an error propagates beyond the boundary of the process context. Even if an error propagates across subsystems, we can recover from the failure if the scope is process-local. Pham *et al.* [11] use SWIFI to evaluate virtualization environments in a cloud infrastructure.

The techniques used in SWIFI are evolving. G-SWFIT precisely emulates general software faults by mutating binary executable code [4]. According to the analysis in [2], G-SWFIT improves the fault injection accuracy. Unfortunately, G-SWFIT does not inject faults that are specific to Linux kernels. So, we use another fault injector that is widely used in the OS community.

Numerous mechanisms for kernel recovery have been proposed to mitigate the impact of kernel failures. Swift *et al.* [13, 12] propose a kernel mechanism for managing and recovering from device driver failures. Otherworld [3] enables us to restart the kernel without discarding the applications' memory states. Phase-based Reboot [14] shortens the downtime involved in a reboot-based recovery.

Table 1: Fault types

Fault types	Description
branch	deletes branches
inverse	flips predictions
ptr	destroys pointers
dstsrc	destroys assignments
interface	omits function arguments
init	omits initialization
irq	deletes restoration of interrupts
off by one	e.g., `ja` change into `jae`
alloc	kmalloc returns NULL
free	deletes kfree
size	makes heap alloc. smaller
bcopy	makes string functions overrun
loop	destroys loop condition
var	allocates huge local valuable
null	omits NULL check

3 Fault Injection

3.1 Fault Injector

We conduct an experimental fault injection campaign in Linux 2.6.38 to investigate the scope of error propagation. The fault injector [8] used in the experiments is widely used in the OS research community [6, 13, 12, 3, 14].

The fault injector obtained from the Nooks web site is ported to the Linux x86 kernel (ver. 2.6.38) and extended to emulate some software bugs that are not included in the original but observed in the Linux empirical bug study [10]. Such extensions include the `size` fault that makes the size of a heap allocation smaller to emulate a heap overrun, the `var` fault that allocates huge local variables, and the `null` fault that emulates missing checks of the null pointers.

In our experiments, 15 types of faults that are injected are listed in Table 1. Due to the space limitations, five faults out of 15, which are peculiar to the kernels, are explained in detail. For ease of understanding, Table 2 lists some examples of the injected faults on the C-language level although the injection is done on the binary level. You can refer to other papers or resources (e.g., [4]) to know the details of other faults because they are common in SWIFI.

Table 2: C-Language Level View of the Injected Faults.

Fault	Before	After
init	`int x = 1;`	`int x;`
irq	`arch_local_irq_restore();`	deleted.
off by one	`while (x < 10)`	`while (x <= 10)`
alloc	`ptr = kmalloc(10, GFP_KERNEL);`	`ptr = NULL;`
free	`kfree(ptr);`	deleted.

- **init**: This fault creates a situation where the initialization of the variables is missed. The instructions responsible for initializing a variable are deleted to create such a situation. More concretely, it deletes an instruction that assigns an immediate value to an address lower than the stack pointer.

- **irq**: A situation is created where a programmer forgets to enable the interrupts after disabling them. The injector removes the calls to `arch_local_irq_restore()`, which restores the disabled interrupts in Linux 2.6.38.

- **off by one**: This fault imitates loop boundary condition errors. The injector changes conditions such as > to >=, < to <=, and so on. For example, "`jae`" is changed into "`ja`".

- **alloc**: This fault makes `kmalloc` return NULL to emulate the shortage of the heap memory. In x86_64, `kmalloc` returns the address of the allocated memory through the `%rax` register. Thus, `call kmalloc` is changed into `xor %rax, %rax` to inject the alloc fault.

- **free**: This fault emulates a situation where the memory is not appropriately released. The injector removes the call to `kfree`, which is responsible for releasing the unused heap memory. Since `kfree` does not return any values, the injector simply deletes the call to `kmalloc`.

The injector rewrites the binary code of the running kernel to inject each type of fault. The code is rewrote at runtime as in other work [5, 11]. The injector disassembles the binary of a randomly selected function in the kernel text segment. Since the faults injected by our injector are context-dependent, it analyzes the disassembled code and searches for proper locations to which each type of fault can be injected. For example, to inject the `var` fault, the injector must change at least two instructions for allocating and deallocating a large local variable.

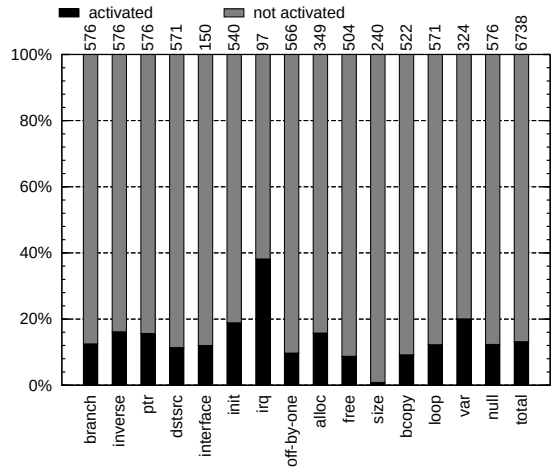
3.2 Error Scope Analysis

A trace of the executed instructions is taken from the fault activation to the error manifestation to keep track of the error propagation. A breakpoint is set in the instruction to which a fault is injected. After the fault is activated, the CPU is set into the single-step execution mode to take a trace of every instruction.

Using the execution trace, the scope of error propagation is analyzed in the same way as a taint analysis. If the injected fault produces an erroneous value, the value is marked as an "error". When the value marked as an "error" is used to calculate another value, the calculated value is also marked as an "error". If the value marked as an "error" is used in the prediction of conditional branches, all the values updated in the taken clause are marked as an "error". If no value marked as an "error" is written to a heap until an oops or panic() is called, the error is concluded to be process-local. Otherwise, the error is concluded to be kernel-global. Our analysis is conservative; i.e., an error is considered kernel-global if we are not confident that it is process-local.

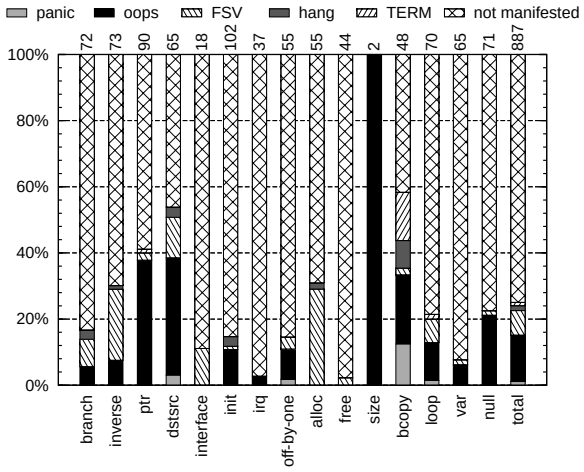
4 Experimental Results

An experimental fault injection campaign has been conducted in Linux 2.6.38 to estimate the reliability after the Linux kernel oops. In this experiment, 6,738 faults are injected to randomly selected locations. We run a workload for each injected fault to activate the faults. We use six benchmarks that all stress the kernel. The six workloads are, 1) UnixBench on ext4, 2) UnixBench on fat, 3) UnixBench on USB, 4) Netperf, 5) Apaly, and 6) Restartd. UnixBench calls a lot of file- and process-related system calls and puts a heavy workload on current file systems. Netperf calls network-related system calls. Aplay invokes sound device drivers. As a benchmark, we listen to a wav file for 10 seconds. Restartd is a benchmark to restart all the system daemons. We run our target system on a VMware workstation to reduce the time for rebooting the kernel after failures. The VMware workstation sometimes detects a critical error in the guest OS and terminates the execution of the guest OS.



(a): **Activated/Not Activated Faults**

This figure shows the relative frequency with which injected faults are activated or not. The number at the end of each bar represents the total number of injected faults.



(b): **Observed Failures**

This figure shows the relative frequency with which activated faults manifest different categories of failures. The number at the end of each bar represents the total number of activated faults.

Figure 1: **Overall Fault Injection Results.**

4.1 Scope of Error Propagation

Figure 1 shows the overall results of our fault injection experiments ("FSV" means fail silence violation and "TERM" means unexpected termination by VMM in Figure 1(b)). There are a total of 6,738 faults injected in our experiments and 13% of the injected faults are activated. Every workload runs 1122 times and 6 faults manifests failures before the workloads start. The kernel oops are called in 14% (124 out of 887) and panic() is called in 1.1% (10 out of 887). 9.9% of the manifested errors do not invoke a kernel oops because they result in fail silence violations, hangs, or unexpected terminations by VMM. 75% of the faults are not manifested.

Since our focus is on the reliability after the kernel oops, we investigate the scope of error propagation in the cases of the 124 kernel oops plus the 10 panic (because most panic() is called by the oops procedure). Figure 2 summarizes the scope of error propagation. According to our experiments, 73% (98 out of 134) of the kernel oops are process-local, while 27% (36 out of 134) of them are kernel-global. This suggests that three quarters of the kernel oops can be recovered simply by revoking the faulty process.

This high rate of process-local errors is attributed to a defensive style of coding in Linux. Linux contains a lot of self-checking codes. For example, `BUG_ON` macro, which is similar to `assert()` in C, checks a given predicate and calls a kernel oops if the predicate is true. Some errors injected by our injector are caught by `BUG_ON` and their propagation is prevented.

A typical example we encountered during the experiments is as follows. The `irq` fault, which removes a call to `arch_local_irq_restore`, which restores disabled interrupts. When this fault is activated, the kernel continues to run with the interrupts disabled. Meanwhile, `lookup_bh_lru(bdev, block, size)` is invoked, which is assumed to be called with the interrupts enabled. It eventually calls `check_irqs_on`, which executes `BUG_ON(irq_disabled())` to check this assumption. Since the interrupts are disabled here (if the

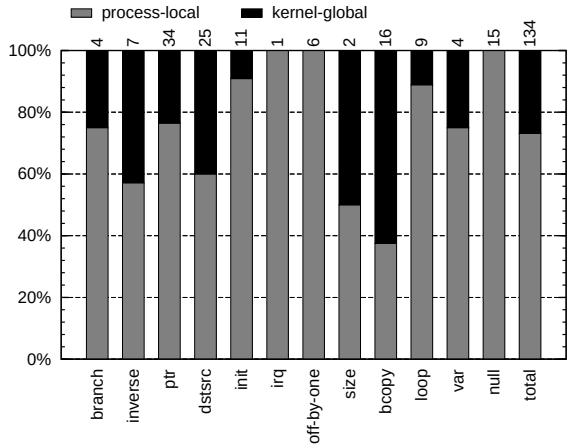


Figure 2: **Scope of Error Propagation**

This figure shows the relative frequency with which propagated errors are process-local or kernel-global. The number at the end of each bar indicates the total number of investigated errors.

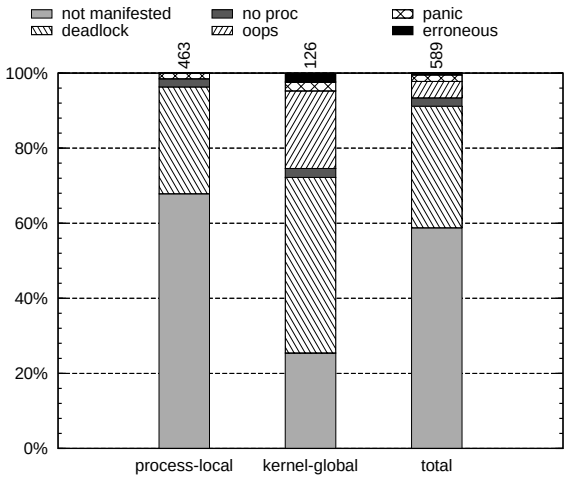


Figure 3: **Kernel behavior after oops**
This figure shows the relative frequency with which the kernel manifests different failure categories after oops recovery. The number at the end of each bar indicates the total number of investigated kernel behaviors.

fault is not injected, the interrupts are enabled here), `BUG_ON` macro detects this incorrect status of interrupts.

Note that our analysis is very conservative. The rate of process-local errors is expected to be higher than 73% in reality. This is because the power of our analyzer of the propagation scope is very limited, and concludes an error is kernel-global if it is not confident that the error is process-local.

4.2 Estimating Reliability after Kernel Oops

According to the results of our analysis on the scope of error propagation, Linux is expected to be reliable with a probability of 73% after the kernel oops. We observe what happens if we run another workload after killing a faulty process on the kernel oops to confirm that the Linux kernel can continue to run after the kernel oops. Note that the kernel cannot continue to run after crashes other than the kernel oops. In reality, it is quite difficult to distinguish process-local errors from kernel-global ones. So, we run a second workload after every kernel oops, regardless of whether it is caused by process-local or kernel-global errors. Just after the kernel oops occur, we also remove the injected fault to analyze the effect of the error propagation caused by it.

Figure 3 shows the summary of the kernel behavior after the kernel oops. The converge of the workloads run after the kernel oops are quite important for precisely estimating the reliability of the Linux kernel. To this end, we inject an identical fault again and again that caused the kernel oops in the previous experiment and run different workloads after the kernel oops. So, the total number of errors is larger in Figure 3 than Figure 2.

No errors manifest in 68% of the process-local errors after the kernel oops. This probability is less than our expectation, where no errors manifest in almost all the cases. Even after the process-local errors, deadlock occurs in 29% (132 out of 463). This is because a faulty process is killed with the lock acquired. Although no global data structures are corrupted in process-local errors, the faulty process holds locks and killing it results in deadlocks after the kernel oops.

In kernel-global errors, no errors manifest in 25% after the kernel oops. This is because the workloads run after the kernel oops do not access the shared data corrupted by the faulty process. When the corrupted data is accessed after the kernel oops, deadlock occurs in most cases. In our experiments, deadlock occurs in 47%. An error inside a critical section tends to result in a failure within the critical section because an error does not usually propagate a long way. Since the accesses to global data structures are controlled by synchronization primitives, the offending process is killed with the lock held and deadlocks are caused afterwards. This behavior of the Linux kernel is preferable because it contributes to fail-stopness after the kernel oops. This result is interesting because no further data corruption occurs even after kernel-global errors in 72% ($= 25\% + 47\%$).

In summary, if we continue to run the Linux kernel after the kernel oops, it runs reliably or stops its execution before trying to access corrupted data with a probability of 91% (not manifested and deadlock in Figure 3). While the kernel compromised by the process-local errors does not always succeed in continuing execution, kernel-global errors do not cause fatal failures in which the operation continues using inconsistent and corrupted data. In other words, the Linux kernel has a good fail-stopness property after the kernel oops.

Killing a faulty process sometimes leads to another problem. `No proc` in Figure 3 indicates cases where workloads running after the kernel oops cannot run as usual because the killed process is mandatory to continue the execution of the workloads. For example, UnixBench on USB cannot be started after kernel oops because a kernel daemon monitoring the plugs for USB devices is killed.

In process-local errors, panic() is called in 1.5% of the cases. It is observed when the kernel detects a buffer overrun in a kernel stack with a canary, or the kernel finds that the faulty contexts are those for interrupts or the init process in the kernel oops procedure. The kernel determines to call panic() regardless of the state of its data structure, and therefore, panic() is observed even when errors are process-local.

In kernel-global errors, oops and panic() are called in

23% of the cases. In these cases, the errors that propagate to global data structures are simple, so access to them can be caught with the kernel oops. Unfortunately, there are three cases (labeled as `erroneous` in Figure 3) in which the Linux kernel continues its operation using inconsistent and corrupted data structures. However, this terrible situation happens only in 0.5% (3 out of 589 errors) of the cases in our experiments.

5 Conclusion

We investigated how reliable Linux is after a kernel oops in this paper. We introduced the concept of the scope of error propagation to investigate the reliability after a kernel oops. The error propagation scope is process-local if an error is confined in the process context that activated it. The scope is *kernel-global* if an error propagates to other processes' contexts or global data structures. Our findings are twofold. First, the error propagation scope is mostly process-local. Thus, Linux remains consistent after a kernel oops in most cases. Second, Linux stops its execution before accessing inconsistent states when kernel-global errors occur because synchronization primitives prevent the inconsistent states from being accessed by other processes.

References

[1] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An Empirical Study of Operating Systems Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Oct. 2001), pp. 73–88.

[2] COTRONEO, D., LANZARO, A., NATELLA, R., AND BARBOSA, R. Experimental analysis of binary-level software fault injection in complex software. In *Proceedings of the IEEE 9th European Dependable Computing Conference (EDCC '12)* (May 2012).

[3] DEPOUTOVITCH, A., AND STUMM, M. Otherworld - Giving Applications a Change to Servive OS Kernel Crashes. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)* (Apr. 2010), pp. 181–194.

[4] DURAES, J., AND MADEIRA, H. S. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering 32*, 11 (Nov. 2006), 849–867.

[5] GU, W., KALBARCZYK, Z., IYER, R. K., AND YANG, Z. Characterization of Linux Kernel Behavior under Errors. In *Proceedings of the 2003 IEEE International Conference on Dependable Systems and Networks (DSN '03)* (Jun. 2003), pp. 459–468.

[6] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Fault isolation for device drivers. In *Proc. of the 39st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09)* (Jun. 2009), pp. 33–42.

[7] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., NORRISH, M., KOLANSKI, R., SEWELL, T., TUCH, H., AND WINWOOD, S. sel4: Formal verification of an os kernel. In *Proceedings of the 22nd ACM symposium on Operating systems principles (SOSP '09)* (2009).

[8] NG, W. T., AND CHEN, P. M. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proceedings of the 29th Symposium on Fault-Tolerant Computing (FTCS '99)* (Jun. 1999), pp. 76–83.

[9] NG, W. T., AND CHEN, P. M. The Design and Verification of the Rio File Cache. *IEEE Transactions on Computers 50*, 4 (Apr. 2001), 322–337.

[10] PALIX, N., THOMAS, G., SAHA, S., CALVÉS, C., LAWALL, J., AND MULLER, G. Faults in Linux: Ten Years Later. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)* (Mar. 2011), pp. 305–318.

[11] PHAM, C., CHEN, D., KALBARCZYK, Z., AND IYER, R. K. Cloudval: A framework for validation of virtualization environment in cloud infrastructure. In *Proc. of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)* (Jun. 2011), pp. 189–196.

[12] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recoverying device drivers. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)* (Dec. 2004), pp. 1–16.

[13] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (Oct. 2003), pp. 207–222.

[14] YAMAKITA, K., YAMADA, H., AND KONO, K. Phase-based Reboot: Reusing Operating System Execution Phases for Cheap Reboot-based Recovery. In *Proc. of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)* (Jun. 2011), pp. 169–180.