Regular Paper

Using Fault Injection to Analyze the Scope of Error Propagation in Linux

Takeshi Yoshimura,† Hiroshi Yamada††,††† and Kenji Kono†,†††

Operating systems (OSes) are crucial for achieving high availability of computer systems. Even if applications running on an operating system are highly available, a bug inside the kernel may result in a failure of the entire software stack. The objective of this study is to gain some insight into the development of the Linux kernel that is more resilient against *software* faults. In particular, this paper investigates the *scope* of error propagation. The propagation scope is *process-local* if the error is confined in the process context that activated it. The scope is *kernel-global* if the error propagates to other processes' contexts or global data structures. The investigation of the scope of error propagation gives us some insight into 1) defensive coding style, 2) reboot-less rejuvenation, and 3) general recovery mechanisms of the Linux kernel. For example, if most errors are process-local, we can rejuvenate the kernel without reboots because the kernel can be recovered simply by killing faulty processes. To investigate the scope of error propagation, we conduct an experimental campaign of fault injection on Linux 2.6.18, using a kernel-level fault injector widely used in the OS community. Our findings are (1) our target kernel (Linux 2.6.18) is coded defensively. This defensive coding style contributes to lower rates of error manifestation and kernel-global errors, (2) the scope of error propagation is mostly process-local in Linux, and (3) global propagation occurs with low probability. Even if an error corrupts a global data structure, other processes merely access to them.

1. Introduction

Operating systems (OSes) are crucial for achieving high availability of computer systems. Kernel-level failures are known to occur less frequently compared with application-level failures, but they have a considerable impact on the overall availability of software systems. Even if applications running on an OS are highly available, bugs inside the kernel may result in a failure of the entire software stack; no application can continue to run on the crashed kernel.

Modern OSes are far from bug-free. Rich functionality of the OSes makes it harder to eliminate all the bugs before shipping. Although the advances in debugging tools, software testing methodologies, static analysis, and formal methods are tremendous, there are many software faults in production-quality OSes. According to Palix et al.[1], the rate of introduction of bugs continues to rise even in Linux 2.6. Our investigation into the change logs of Linux 2.6.24 and 2.6.25 also reveals that there are critical bugs inside the kernel core components.

The objective of this study is to gain some insight into design of the Linux kernel that is resilient against *software* faults. To this end, it is critically important to understand Linux kernel behaviors under software faults. We introduce the concept of the *scope of error propagation*. The propagation scope is *process-local* if the error is confined in the process context that activated it. The scope is *kernel-global* if the error propagates to other processes' contexts or global data structures. To the best of our knowledge, no existing work investigated the scope of error propagation.

This distinction between process-local and kernel-global errors is significant. If most errors are process-local, the kernel can recover from most errors simply by killing and revoking the resources of the faulty process. This implies that the Linux kernel can be partially rejuvenated without rebooting the entire OS; it can restart the killed process when the fault is transient (e.g., timing bugs). If most errors are kernel-global, the recovery becomes hopeless because corrupted global data structures must be recovered to continue processing. In this case, a mechanism isolating propagated errors should be developed rather than recovery mechanisms.

To investigate the scope of error propagation, a series of fault injection experiments is conducted. The fault injector used in our experiments is the existing one[2] that is widely used

† Keio University
†† Tokyo University of Agriculture and Technology
††† JST CREST

to evaluate the OS dependability in the OS research community[3)~7)]. It focuses on the emulation of low- and high-level software faults, including ones specific to OS kernels.

The distinguished feature of our study is threefold. First, we investigate the "scope" of error propagation; an error caused by a software fault propagates outside or confines within the context of the faulty process. Second, we focus on low- and high-level software faults, while the primary target of previous work[8)~10)] is low-level hardware faults such as flipping bits in memory. Flipping bits in memory can indirectly emulate low-level programming faults, however, high-level programming faults such as argument faults are neither emulated directly nor considered enough. Third, we show the detailed analysis of faults that are activated but do not manifest themselves.

In our fault injection experiments, 864 faults are injected into Linux 2.6.18 and 20% of the injected faults are activated. Error propagation is investigated by using a built-in kernel debugger. The major findings include:

- Our target kernel (Linux 2.6.18) is coded in a defensive way. It frequently checks the integrity of function arguments, return values, and other important variables. This style of defensive coding contributes to lower rates of error manifestation (36% of the fault activation) and kernel-global errors (16% of the failures).
- The scope of error propagation is mostly process-local in Linux (84% of the failures). This implies that the Linux can be rejuvenated without reboots with high probability. 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 a faulty process.
- Global propagation of errors occurs with low probability. Interestingly, even if a global data structure is corrupted, the corrupted data cannot be accessed from the other processes in our experiment. This is because the faulty process crashes with a lock acquired inside a critical section.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 explains the software fault injector we used. Section 4 describes our methodology of the experimental campaign of fault injection. Section 5 reports our experimental results. Section 6 discusses the directions towards more resilient structure of the Linux kernel. Section 7 concludes this paper.

2. Related Work

Understanding the kernel behavior under fault manifestation can be an aid for kernel developers to improve the kernel dependability or develop the mechanisms for kernel recoveries. OS kernel behavior under fault manifestation has been widely examined.

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. This work focuses on the "scope" of error propagation (i.e., process-local or kernel-global), while previous work focuses on other aspects of error propagation than the scope of error propagation. This work is extended from our previous work[11),12)] and reports more detailed results.

Gu *et al.* [8)] 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. Pham *et al.*[13)] use SWIFI for their framework that automates to validate the robustness of virtualized environments based on KVM or Xen hypervisors.

Chen *et al.*[10)] and another paper from Gu *et al.*[9)] investigate behavioral difference caused by different combinations of CPU architectures and OSes (five combinations of CPU and OSes are investigated in total). These studies indicate a good insight into the design principles of CPU architectures and OSes that are resilient to faults. However, these studies do not address the scope of error propagation. The fault models considered in these studies use device-level transient faults, while the fault model used in our study is low- and high-level programming errors.

The techniques used in SWIFI are evolving. G-SWFIT precisely emulates general software faults by mutating binary executable code[14)]. According to the analysis by Controneo *et al.*[15)], G-SWFIT improves the fault injection accuracy. Unfortunately, G-SWFIT does not emulate faults that are specific to Linux kernels. So, we use another fault injector that is widely used in the OS community.

Aside from fault injection studies, software bugs in the production-quality OSes such as Linux are extensively examined. Chou *et al.*[16] apply a static analyzer to Linux versions 1.0 through Linux 2.4.1 to study the trend of software bugs in the Linux kernels. Palix *et al.*[1] are the most recent follow-up that investigates Linux versions 2.6.0 to 2.6.33. The primary goal of these studies is to identify the distribution and lifetime of certain kinds of faults in the Linux kernels.

To mitigate the impact of kernel failures, numerous mechanisms for kernel recovery have been proposed. Swift *et al.*[3),4)] propose a kernel mechanism of managing and recovering from device driver failures. Otherworld[5)] restarts the kernel without discarding applications memory states when the kernel crashes. Phase-based Reboot[6)] shortens downtime involved in reboot-based recovery.

3. Fault Injector

We investigate the scope of error propagation in a commodity OS kernel (i.e., Linux) under software fault manifestation. To this end, an experimental campaign of fault injection is conducted in Linux to examine how it reacts to the injected faults. The injector that is originally obtained from the Nooks web site is ported to the x86 Linux 2.6.18 kernel. This section briefly describes the injector and the fault types it injects.

3.1 Overview

The injector[2)] emulates low- and high-level programming bugs specific to OS kernels. It changes individual instructions in the kernel text segment. These faults are intended to approximate the assembly-level manifestation of real C-level programming errors. For example, the injector emulates missing initialization by deleting instructions that are responsible for variable initialization. The details of the faults are described in Section 3.2.

The injector is widely used to evaluate and validate recovery mechanisms in the OS research community. For example, it was used to evaluate the fault tolerance of the file system cache[2)], recovery mechanisms for device drivers[3),4)], a kernel mechanism for applications to survive OS crashes[5)], and a quick mechanism for reboot-based recovery[6)].

The injector runs in the kernel and provides a system call interface to specify the parameters of fault injection. It rewrites the binary code of the running kernel to inject each type of fault. 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.

3.2 Injected Faults

The injector emulates 10 types of faults. These faults range from low-level hardware faults to high-level software faults. Since our primary concern is in programming errors, we omitted 3 types of faults that emulate low-level hardware faults. So, 7 types of faults are injected in our experiments. For ease of understanding, Table 1 lists some examples of injected faults at the C-language level, although the injection is done at the binary level.

- **INIT FAULT**: INIT fault creates a situation where the initialization of variables is missed. To create such a situation, the injector deletes instructions responsible for initializing a variable by copying a constant value. More concretely, it deletes an instruction that assigns an immediate value to the address lower than the stack pointer.
- **DST&SRC FAULT**: This fault corrupts assignment statements. This creates a situation where the assignment is incorrect due to a programming error. To do this, the injector corrupts the value of the source or the destination by flipping the bits of the value.
- **PTR FAULT**: This fault emulates pointer corruption by corrupting the addressing bytes of instructions. The injector either flips a bit within the addressing-form specifier byte (ModR/M) or the scale, index or base (SIB) byte following the instruction opcode.
- **BRANCH FAULT**: This fault emulates an incorrect control flow by deleting a jump instruction involved in the conditional statement. By doing this, the injector emulates branch errors and error handling faults.
- **INVERSE FAULT**: The injector also reverses the predicates of conditional statements to inject incorrect control flows. For example, this fault changes "`je`" into "`jne`" to reverse the predicate.
- **INTERFACE FAULT**: This fault corrupts one of the arguments passed to a procedure. To create this situation, the in-

Table 1 C-Language Level View of the Injected Software Faults.
This table shows examples of the injected faults at the C-language level.

Fault	Before	After
INIT	`int x = 0;`	`int x;`
DST&SRC	`x += 1;`	`x += 2;`
PTR	`ptr = list->prev;`	`ptr = list->next;`
BRANCH	`if (x != 0) return;`	`return;`
INVERSE	`if (x == 0)`	`if (x != 0)`
INTERFACE	`func(1, 2, 3);`	`func(1, 214, 3);`
IRQ	`local_irq_restore();`	deleted.

jector deletes an instruction that copies a value at an address below the base pointer to registers or memory. For example, the injector can change the call `foo(a, b)` to `foo(X, b)`, where `X` is a corrupted value, by deleting the instruction that copies `a` to a register or memory.

- **IRQ FAULT**: When an IRQ fault is injected, the injector creates a situation where a kernel developer forgets to enable interrupts after disabling them. The injector removes `local_irq_restore()` calls. When a call to `local_irq_restore()` is removed, the interrupt mask is not restored and thus the disabled interrupts continue to be disabled.

4. Methodology

To investigate the scope of error propagation, we track the Linux kernel behavior when an injected fault is activated. The kernel version of Linux we use is 2.6.18.8. 864 faults are injected in our experiments. To track how the Linux kernel reacts to the injected faults, we take the following steps:

(1) *Injecting a fault:* We request the injector to inject a fault. In our experiments, only the text segment is modified to inject faults as our target is programming errors. This does not imply no data is corrupted in our experiments. Data in heap or stack may be corrupted by the injected erroneous instructions. To trace the kernel execution after the fault is activated, we set a breakpoint at the instruction to which a fault is injected. When the breakpoint is hit, the control is transferred to KDB, a built-in kernel debugger for Linux. We do not inject faults into the KDB code.

(2) *Running a workload:* The workload that we use to activate injected faults is to restart all the daemons. Since the daemons extensively issue system calls, the kernel code runs very frequently while the daemons are restarted.

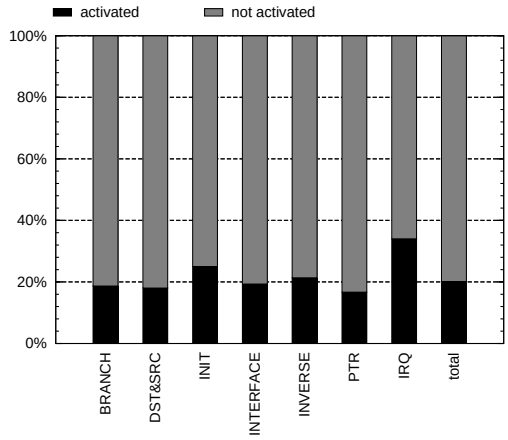
(3) *Tracing error propagation:* After the fault is activated, the CPU is set into the single-step execution mode to take a trace of every instruction. Using the execution trace, the scope of error propagation is analyzed in the same way as taint analysis. If the injected fault produces an erroneous 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, the error is concluded to be process-local. Otherwise, the error is concluded to be kernel-global. The kernel execution is tracked until kernel failures (e.g., kernel panic). If all the daemons are restarted successfully, the error is classified into "not manifested".

Note that error propagation is investigated at the assembly code level in our experiments, although this section describes the analysis of error propagation at the source code level for readability. Error propagation can be analyzed more precisely if it is analyzed at the assembly level. For example, compilers generate optimized code that shares common expressions. Supposed that there are two expressions: x = a + b and y = (a + b) * c. If a fault is injected into the former a + b, it propagates to the latter.

5. Experiments

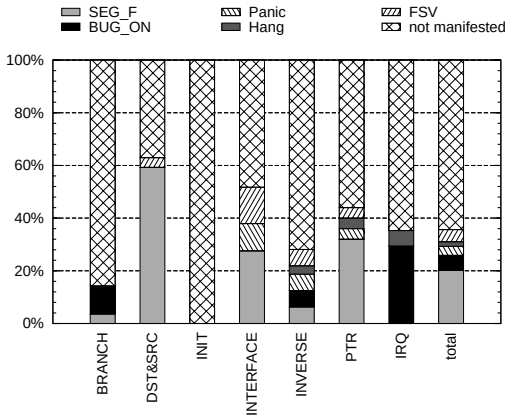
5.1 Experimental Setup

Our experimental campaign of fault injection is carried out on VMWare Workstation 7.1.2 running on Windows 7. We run Fedora 8 (Linux 2.6.18.8) in a guest virtual machine that consists of 1 CPU, 1 GB of memory and 20 GB hard disk drive. The host CPU is 2.53 GB Core2 Extreme CPU. The kernel configuration



(a) Activated/Not Activated Faults

This figure shows the relative frequency with which injected faults are activated or not.



(b) Observed Failures

This figure shows the relative frequency of not-manifested errors and the failure categories of manifested errors

Fig. 1 Overall Fault Injection Results.

is default. Note that the failures encountered in these experiments are triggered by injected faults, not bugs in the Linux kernel, although real bugs inside the kernel can trigger failures during our experiments.

5.2 Overall Results

Figure 1 shows the overall results of our fault injection experiments. In total, 864 faults are injected and 20% of them are activated. Figure 1(b) shows the failures which are observed after the fault activations. Segmentation failures ("SEG_F" in Figure 1(b)) are caused in 20% of the fault activations. They occur when the kernel attempts to access illegal pages. Intentional kernel crashes caused by `BUG_ON` are observed in 6% ("`BUG_ON` in Figure 1(b)). `BUG_ON` denotes a situation where Linux `BUG_ON` macro, similar to C assert, detects an erroneous state in the kernel. The other failures are panic, hangs and fail silence violations ("FSV" in Figure 1(b)). 64% of the activated faults do not manifest themselves.

Figure 2(a) summarizes the result of the scope analysis. 84% of the manifested errors are process-local, while 16% of them are kernel-global. This suggests that 84% of the kernel failures can be recovered simply by revoking the faulty process.

Figure 2(b) summarizes observed failures in terms of their error propagation scope. These segmentation failures occur in both propagation scopes with the highest probability out of all the observed failures (56% of all the manifested errors). All of the fail silence violations and `BUG_ON` are caused only by process-local errors. This result implies that `BUG_ON` effectively prevents global propagation in the kernel as described in Section 5.3.1.

5.3 Scope Analysis

This section shows the detailed analysis of kernel traces. Error propagation are thoroughly examined in terms of their scope.

5.3.1 Process-local errors

Table 2 shows typical examples of each failure type caused by process-local errors. The table lists an injected fault type, a memory address where the fault is injected, the location at the source code level, and the instructions and C-code before/after the fault injection.

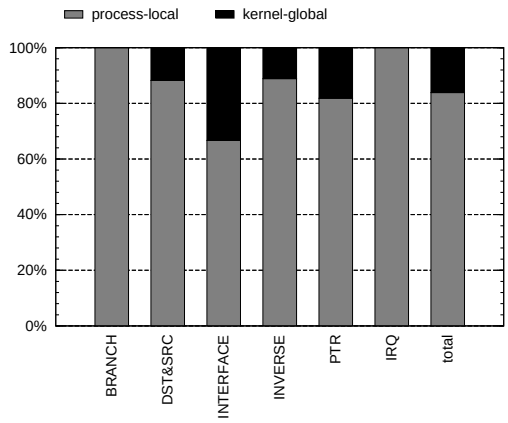
(a) *Segmentation Failure:* As shown in Figure 2(b), 56% of the process-local errors lead to segmentation failures. Table 2(a) shows the detail of a typical case that leads to a segmentation failure. In this case, a null pointer is passed to a function that expects the passed pointer not to be null. This fault is injected by INVERSE FAULT. More concretely, the code

```
if (sd->s_iattr) {
    set_inode_attr(inode, sd->s_iattr);
    ...
```

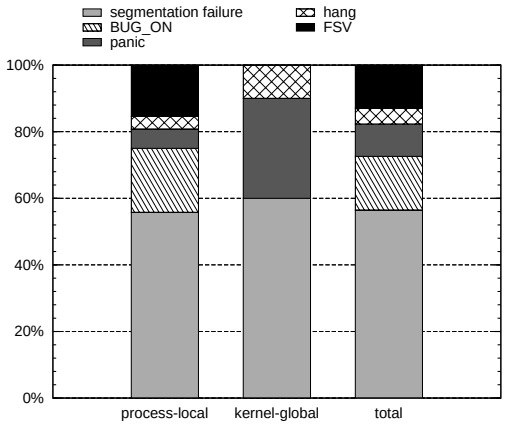
is modified to

```
if (!sd->s_iattr) { // FAULT injected here
    set_inode_attr(inode, sd->s_iattr);
    ...
```

In the original code, `set_inode_attr` is called only when `sd->s_iattr` is not NULL. However, `set_inode_attr` is called when `sd->s_iattr` is NULL in the modified code. As



(a) Error Propagation Scope
This figure shows the relative frequency of process-local or kernel-global errors.



(b) Failure Type by Scope
This figure shows the relative frequency with which the kernel causes different failure categories after fault activations.

Fig. 2 Overall Result of Scope Analysis

a result, parameter `iattr` in `set_inode_attr` becomes NULL as shown below. The dereference of `iattr` causes a segmentation fault.

```
void set_inode_attr(inode, iattr)
{
    // Failure
    inode->i_mode = iattr->ia_mode; // iattr is NULL
```

In this case, a null pointer is passed across function calls but no global data structures are updated with the incorrect null pointer. Thus, the scope of error propagation is process-local.

The following is short descriptions of the other typical process-local errors leading to segmentation failures. First, a function that is expected to return a non-null pointer to the caller is modified to return a null pointer. As a result, the caller crashes because it dereferences the returned null pointer with no checks. Second, a pointer is initialized incorrectly and dereferenced later in the same function. Finally, a return address stored in a stack is destroyed. So, a segmentation fault occurs when this function returns. All of these errors do not propagate outside the contexts of faulty processes.

(b) *BUG_ON:* As shown in Figure 2(b), 19% of the process-local errors lead to `BUG_ON`. An example of this failure is caused by IRQ FAULT, which removes the call to `local_irq_restore` to forget to enable disabled interrupts. After this fault is activated, the kernel continues to run with the interrupts disabled. Meanwhile, `lookup_bh_lru(bdev, block, size)` is invoked. This function eventually calls `check_irqs_on`, which executes `BUG_ON(irq_disabled())`. Since the interrupts are disable here (if the fault is not injected, the interrupts are enable here), `BUG_ON` macro successfully detects this incorrect status of interrupts.

This experimental result suggests that `BUG_ON` macro is effective to prevent global error propagation. If `BUG_ON` is not used to check the status of interrupts, blocking functions are called with the interrupts disabled and thus, the deadlock or other serious situations would be caused. In the current versions of Linux, `BUG_ON` macro is inserted manually according to the developers' experiences and intuitions. We expect that more systematic methods are required in order to help the developers insert `BUG_ON` macros correctly.

(c) *Panic:* As shown in Figure 2(b), 6% of the process-local errors cause kernel panic. Table 2(c) shows a typical example of panic. In this case, a fault is injected into an interrupt handler. More concretely, an argument to function `neigh_update` is corrupted and thus the address of `neigh->dev`, which is calculated from the corrupted argument, becomes an incorrect value. As a result, the first access to `neigh->dev` causes a segmentation failure. Since this code is executed in an interrupt handler, the kernel invokes `panic` instead of causing a segmentation failure.

(d) *Fail silence violation:* There are 15% of the process-local errors that lead to fail silence

Table 2 Faults causing process-local errors

(a) Segmentation Failure

Fault	INVERSE FAULT
Memory Address	`sysfs_new_inode+0x5c`
Code Location	`fs/sysfs/inode.c, line:134`
Original Instruction	`je sysfs_new_inode+0x97`
Modified Instruction	`jne sysfs_new_inode+0x97`
Original Code	`if (sd->s_iattr) {`
Modified Code	`if (!sd->s_iattr) {`

(b) BUG_ON

Fault	IRQ FAULT
Memory Address	`kfree+0x5f`
Code Location	`mm/slab.c line: 3463`
Original Instruction	`push %esi popf`
Modified nstruction	`nop nop`
Original Code	`local_irq_restore(flags);`
Modified Code	deleted

(c) Panic

Fault	INTERFACE FAULT
Memory Address	`neigh_update+0x1ed`
Code Location	`net/core/neighbour.c line:894-895`
Original Instruction	`mov 0xc(%ebp), %eax`
Modified Instruction	`nop nop nop`
Original Code	`void (*update)(...) = neigh->dev-> header_cache_update;`
Modified Code	`void (*update)(...) = (struct netdevice *)(0x6)-> header_cache_update;`

(d) Fail silence violation

Fault	SRC&DST FAULT
Memory Address	`sock_alloc_fd+0xb`
Code Location	`net/socket.c, line:380`
Original Instruction	`mov %eax, %ebx`
Modified Instruction	`mov %esp, %ebx`
Original Code	`fd = get_unused_fd();`
Modified Code	`get_unused_fd();`

(e) Hang

Fault	IRQ FAULT
Memory Address	`do_softirq+0x48`
Code Location	`kernel/softirq.c, line:215`
Original Instruction	`push %esi popf`
Modified Instruction	`nop nop`
Original Code	`local_irq_restore(flags);`
Modified Code	deleted

violation as shown in Figure 2(b). In our experiments, Fail silence violations often derive from kernel error detections. Despite their correctness, the kernel starts to handle the detected errors by the usual error processing manner. Besides, such error processing tends to simply abandon the current processing and return a corresponding erroneous value (e.g., `EINVAL`), therefore, global data structures are merely updated before fail silence violations occur. In our experiments, we do not observe any kernel-global errors that lead to fail silence violations.

The following is a typical example of fail silence violation. In this example, the injected fault generates a situation in which there is no unused network sockets. So, the Linux kernel considers no network sockets can be created. The following is simplified code for explanation. The original code

```
int sock_alloc_fd(...) {
    int fd;
    fd = get_unused_fd(); // Fault injected here
    ...
    return fd;
}
```

is modified to

```
int sock_alloc_fd(...) {
    int fd;
    get_unused_fd(); // "fd =" is removed
    ...
    return fd;
}
```

In the modified code, `fd` is not initialized. In our experiment, uninitialized `fd` happens to be negative. As a result, `sock_alloc_fd` returns a negative value to its caller. The caller is:

```
// sock_alloc_fd is called here
// retval becomes negative
retval = sock_alloc_fd(sock);
// Linux considers no socket
// can be allocated
if (retval < 0)
    goto out_release;
...
out_release:
// socket is released and
// a negative value is returned
sock_release(sock);
return retval;
```

In the above code, the Linux kernel considers there is no room to create a new socket because `sock_alloc_fd` returns a negative value. As a result, a process cannot create a new socket even though there is enough room to create new sockets.

(e) *Hang:* As shown in Figure 2(b), there are 5% cases in which the Linux kernel hangs up. The typical example is shown in Table 2(e). In this example, IRQ FAULT is injected into `do_softirq` which schedules pending software interrupts. When `do_softirq` returns, the kernel hangs immediately without dumping the stack trace. So, we cannot trace the kernel behavior using KDB. Since we can not determine from the source code which function is executed after `do_softirq` returns, further information cannot be obtained in this case.

Table 3 A Kernel-global error

Fault	INTERFACE FAULT
Memory Address	`rb_erase+0x1e9`
Function	`lib/rbtree.c, line:178`
Original Instruction	`mov 0x0(%ebp),%ebx`
Modified Instruction	`nop nop nop`
Original Code	`node = root->rb_node;`
Modified Code	`node = parent->rb_right;`

5.3.2 Kernel-global errors

16% of the errors are kernel-global as shown in Figure 2(a), while all the other errors are process-local. Some of the process-local errors propagate across multiple function calls but the propagations are limited to function arguments, return values, and local variables. This is probably because global data structures, shared among multiple processes, are used to store stable, consistent states rather than transient, temporary states. Experienced programmers like Linux developers write defensive code that checks data integrity and/or confirms the assumptions on function arguments. A data is checked again and again before it is written to global data structures.

Table 3 shows the detail of a representative kernel-global error. In this case, a fault is injected into a function that manages red-black trees, a type of self-balancing binary search tree, used for storing sortable key-value pairs. More specifically, INTERFACE FAULT is injected into the call to `__rb_erase_color`. Function `__rb_erase_color` takes three arguments: `node`, `parent`, and `root` whose types are all `struct rb_node*`. By the INTERFACE FAULT, argument `node` that should be `root->rb_node` is modified to `parent->rb_right`. As you can imagine from the arguments, `__rb_erase_color` manipulates tree structures in the heap. The incorrect argument leads to the corruption of the global tree structures. When the kernel traverses a broken red-black tree, it crashes due to segmentation fault. Since global data structures are corrupted by injected faults, the scope of this error is kernel-global.

There is one important thing to be noted. The fault shown in Table 3 corrupts global data structures, however, this error is *never* propagated to other processes than the faulty one. This is because the faulty process holds a lock (more precisely, semaphore) for exclusive access to global data structures. When a faulty process causes a segmentation fault, it does not release the lock. Since the other processes cannot acquire the lock, they cannot access to the broken data structures; the corrupted data is never propagated to the other processes.

Table 4 Summary of Not-Manifested Errors. *The table shows the number of errors for each reason that activated errors do not manifest themselves. We conclude that an error does not manifest itself when one of these situation is observed during the tracing of error propagation. The untraceable cases are discussed in detail.*

Reason	# of errors
Corrected	8
Not affecting	10
Error processing omitted	18
Incorrect warning	4
Almost correct operation	15
Aging	6
Lucky	40
Untraceable	11
Total	112

5.4 Not-Manifested Errors

To understand Linux behaviors under software faults, it is critically important to analyze the reason why activated faults do not manifest themselves. As pointed out in many literatures, activated faults do not always manifest themselves. In our campaign of fault injection, These "not-manifested" errors are observed in 64% of the fault activations. If an error is corrected during the execution, the analysis aids in proposing defensive coding styles effective for kernels.

In our experiments, we trace error propagation with the kernel debugger until the kernel detects an error or we are sure of the error not manifesting itself. Table 4 shows the summary of the errors not manifested in our experiments. In this table, these errors are classified into 8 cases, based on the reason why they do not manifest themselves.

Corrected: "Corrected" indicates a situation in which an erroneous state is corrected by the Linux kernel. A typical example of this error is as follows. As shown in Table 5, a fault is injected to remove the initialization of `oldpolicy`. In the original code, `oldpolicy` is initialized to `-1`. This error is corrected as follows.

```
int oldpolicy; // should be initialized to -1
...
if (unlikely(oldpolicy != -1 ....)) {
    policy = oldpolicy = -1; // error corrected
```

Not affecting: "Not affecting" indicates a situation where an erroneous state is not used

Table 5 Examples of Not-Manifested Errors

(a) Corrected

Fault	INIT FAULT
Memory Address	`sched_setscheduler+0x44`
Code Location	`kernel/shed.c, line:4087`
Original Instruction	`movl $0xffffffff, 0xfffffffec(%ebp)`
Modified Instruction	`nop nop ... nop`
Original Code	`int oldpolicy = -1;`
Modified Code	`int oldpolicy;`

(b) Not affecting

Fault	INIT FAULT
Memory Address	`rebalance_tick+0xda`
Code Location	`kernel/sched.c line: 2530`
Original Instruction	`movl $0x0, 0xfffffff0(%ebp)`
Modified nstruction	`nop nop ... nop`
Original Code	`int all_pinned = 0;`
Modified Code	`int all_pinned;`

(c) Error processing omitted

Fault	BRANCH FAULT
Memory Address	`follow_page+0xd8`
Code Location	`mm/memory.c, line:935`
Original Instruction	`je follow_page+0x1aa`
Modified Instruction	`nop nop ... nop`
Original Code	`if (!ptep) goto out;`
Modified Code	deleted

(d) Incorrect warning

Fault	BRANCH FAULT
Memory Address	`net_tx_action+0x37`
Code Location	`kernel/sched.c, line:2845`
Original Instruction	`je net_tx_action+0x55`
Modified Instruction	`nop nop`
Original Code	`if (unlikely(!(x))) {`
Modified Code	deleted

(e) Almost correction operation

Fault	INIT FAULT
Memory Address	`schedule+0xd2`
Code Location	`kernel/sched.c, line:3341`
Original Instruction	`movl 0x3b9aca99, 0xffffffc4(%ebp)`
Modified Instruction	`nop nop ... nop`
Original Code	`run_time = NS_MAX_SLEEP_AVG;`
Modified Code	deleted

by the kernel. For example, a local variable is corrupted but not used at all until the end of the function after the injection, as described in Table 5(b). In this example, local variable `all_pinned`, which is not initialized, is not used in our experiments until the function returns.

Error processing omitted: "Error processing omitted" indicates a situation where the code for error processing is omitted. This error does not manifest itself during the experiments unless the omitted error processing becomes necessary. The detail of a typical example of this case is shown in Table 5(c).

Incorrect warning: "Incorrect warning" indicates a situation where warning messages are displayed even though those messages should not be displayed. This is caused by the omission of conditional jumps that judge if warning messages should be displayed. The detail is shown in Table 5(d).

Almost correction operation: "Almost correction operation" indicates a situation where the kernel behavior is slightly changed from the expected one but the kernel continues to run as normal. Most of these errors are related to scheduling parameters that affect the scheduling behavior of the kernel. In the example shown in Table 5(e), the code for initializing local variable `run_time` is removed by fault injection. Since `run_time` is used to calculate the sleeping time of processes, it changes the scheduling behavior if set improperly. As shown below, even if `run_time` becomes erroneously large, the kernel code corrects the value. As a result, the kernel continues to run almost normally.

```
...
// Following statement removed
run_time = NS_MAX_SLEEP_AVE;
...
// prev->sleep_avg becomes incorrect here
prev->sleep_avg -= run_time;

// prev->sleep_avg corrected if necessary
if ((long)prev->sleep_avg <= 0)
    prev->sleep_avg = 0;
```

Aging: "Aging" indicates a situation where resource leakage occurs. Software aging is a serious problem but the aging errors seem not to manifest themselves during the short duration of fault injection experiments.

Lucky: "Lucky" indicates a situation where an error is activated but happens to cause nothing wrong. For example, INIT FAULT removes code for initializing a local variable to zero, whose value happens to be zero. Another example is from `tty_register_driver`, which is used to register a new major device. A PTR FAULT is injected into this function. In this case, the major device number of the new device becomes an unexpected number but the operation itself continues normally.

Untraceable: There are 11 cases in which we cannot trace error propagation completely. The faults are injected into the code for the socket management, and corrupt packet headers to be sent out to network. The actual operations of sending out the packets are performed asyn-

chronously. So, we cannot trace the sending-out operations with the kernel debugger. We carefully observe the network behavior of the target machine but notice nothing in particular. This is probably because the packets with incorrect headers are destroyed somewhere deeper in network drivers. As a result, this type of errors does not manifest themselves.

6. Discussion

Our findings through the experiments are threefold. First, the Linux kernel is coded in a defensive way. This means that the Linux kernel frequently checks the integrity of function arguments, return values, and other important variables. This defensive coding style would have been introduced to ease debugging and diagnosis of failures. A typical example of the defensive coding in Linux is the use of `BUG_ON` macro, which checks the integrity of the kernel internal states. The use of `BUG_ON` aids in early error detections to prevent error propagation over the entire kernel. One interesting direction towards more resilient Linux is to develop a systematic method that determines the locations where `BUG_ON` macros are inserted and conditions given to those macros. Current static analysis tools are expected to give invaluable hints on the locations and conditions of `BUG_ON` macros.

Second, the scope of error propagation is mostly process-local in Linux. As you can see from our experimental results, most of the activated faults are process-local and do not propagate outside the contexts of faulty processes. This implies that the Linux can be rejuvenated without reboots with high probability. If an error does not propagate outside the context of the faulty process, the kernel states (including global data structures and other processes' contexts) are consistent. Thus, we can recover a consistent kernel state simply by revoking the context of the faulty process.

From our experiments, we have learned that our definition of "process-local" and "kernel-global" is somewhat ambiguous and there is room for further discussion. For example, some errors that cause software aging can be viewed as kernel-global because a resource leakage of a process affects all the other processes in the system. On the other hand, those errors can be viewed as process-local because no global data structures are corrupted; all processes are viewing consistent image of global data structures.

Finally, the global propagation of errors occurs in lower rates (16% of the failures). This is probably due to the nature of the defensive programming style in the Linux kernel. Data integrity is checked again and again before the data is written to global data structures.

One interesting finding in our investigation is that even if a global data structure is corrupted, the corrupted data cannot be accessed from the processes other than the faulty process. This is because global data structures are usually protected with an exclusive lock to avoid concurrent access to them. When a faulty process corrupts a global data structure, it often causes a segmentation fault before it exits the critical section. As a result, no other processes can access to the corrupted data structures.

This finding suggests that a new style of defensive programming. If a faulty process is revoked with a lock acquired, other processes cannot proceed because they cannot acquire the lock. To avoid this, when a faulty process with some locks acquired is revoked, the kernel should release the locks. Note that this design of the kernel increases the possibility that an error is propagated outside the faulty process context through corrupted global data structures. To prevent other processes from accessing corrupted data structures, when a process enters a critical section previously locked by a faulty process, it should check the integrity of the global data structures. If the integrity is confirmed, the process can proceed to access the data structures. Otherwise, the process tries to cure the corrupted data structures. If it succeeds, the process can proceed normally. If it fails, the process gives up accessing the data and calls `panic` to crash the kernel.

7. Conclusion

This paper investigates the Linux behavior under software faults. Our objective of this study is to gain some insight into 1) defensive coding style, 2) reboot-less rejuvenation, and 3) general recovery mechanisms of the Linux kernel. In particular, this paper focuses on the analysis on the *scope* of error propagation. If an error propagates inside the context of the faulty process, it is called process-local. If an error propagates outside the context of the faulty process, it is called kernel-global. To this end, we conduct an experimental campaign of fault injection on Linux. Since our focus is on software faults (in other words, software bugs),

we use an existing software fault injector especially designed for injecting kernel-level software faults. It injects low- and high-level software faults. It is widely used in the OS research community. The major findings include:

- The Linux kernel is coded in a defensive way. It frequently checks the integrity of function arguments, return values, and other important variables. This style of coding contributes the resilience to activated faults. In particular, it contributes to lower rates of error manifestation (36% of the fault activation) and global propagation (16% of the failures). In our experiments, activated errors are often corrected or mitigated to avoid serious failures. This is because the Linux kernel checks data integrity again and again before updating global data structures.
- The scope of error propagation is mostly process-local in Linux. This implies that Linux can be rejuvenated without reboots with high probability. If an error does not propagate outside the context of a faulty process, time-consuming reboots can be avoided since we can recover from the failure simply by killing the faulty process.
- Global propagation of errors occurs with low probability. Interestingly, even if a global data structure is corrupted, the corrupted data cannot be accessed from the other processes if the faulty process is killed within a critical section.

We believe our results of experimental fault injection suggest many directions of further research. As discussed in Section 6, we can take various approaches to improve the dependability of the Linux kernel. First, we expect that a tool for effectively inserting `BUG_ON` macros are required. Second, the Linux kernel can be rejuvenated without reboots with high probability. A mechanism that distinguishes a situation that can be recovered without reboots needs to be developed. Finally, the kernel can be recovered from kernel-global errors if we develop a sophisticated mechanism of handling errors in critical sections.

References

1) Palix, N., Thomas, G., Saha, S., Calvés, C., Lawall, J. and Muller, G.: Faults in Linux: Ten Years Later, *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pp. 305–318 (2011).
2) Ng, W. T. and Chen, P. M.: The Systematic Improvement of Fault Tolerance in the Rio File Cache, *Proceedings of the 29th Symposium on Fault-Tolerant Computing (FTCS '99)*, pp. 76–83 (1999).
3) Swift, M. M., Bershad, B. N. and Levy, H. M.: Improving the Reliability of Commodity Operating Systems, *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp. 207–222 (2003).
4) Swift, M. M., Annamalai, M., Bershad, B. N. and Levy, H. M.: Recoverying Device Drivers, *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp. 1–16 (2004).
5) Depoutovitch, A. and Stumm, M.: Otherworld - Giving Applications a Change to Servive OS Kernel Crashes, *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, pp. 181–194 (2010).
6) Yamakita, K., Yamada, H. and Kono, K.: Phase-based Reboot: Reusing Operating System Execution Phases for Cheap Reboot-based Recovery, *Proc. of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)*, pp. 169–180 (2011).
7) Ng, W. T. and Chen, P. M.: The Design and Verification of the Rio File Cache, *IEEE Transactions on Computers*, Vol. 50, No. 4, pp. 322–337 (2001).
8) Gu, W., Kalbarczyk, Z., Iyer, R. K. and Yang, Z.: Characterization of Linux Kernel Behavior under Errors, *Proceedings of the 2003 IEEE International Conference on Dependable Systems and Networks (DSN '03)*, pp. 459–468 (2003).
9) Gu, W., Kalbarczyk, Z. and Iyer, R. K.: Error Sensitivity of the Linux kernel Executing on PowerPC G4 and Pentium 4 Processors, *Proc. the 4th IEEE International Conference on Dependable Systems and networks (DSN '04)*, pp. 887–896 (2004).
10) Chen, D., Jacques-Silva, G. and Mealey, B.: Error Behavior Comparison of Multiple Compuing System: A Case Study Ui Linux on Pentium, Solaris on SPARC, and AIX and POWER, *Proc. of the 14th IEEE Pacific Rim International Symposium On Dependable Computing (PRDC '08)*, pp. 339–346 (2008).
11) Yoshimura, T., Yamada, H. and Kono, K.: Can Linux be Rejuvenated without Reboots?, *Proceedings of the IEEE 3rd International Workshop on Software Aging and Rejuvenation (WoSAR '11)* (2011).
12) Yoshimura, T., Yamada, H. and Kono, K.: Is Linux Kernel Oops Useful Or Not?, *Proceedings*

of the 8th Workshop on Hot Topics in System Dependability (HotDep'12) (2012).
13) Pham, C., Chen, D., Kalbarczyk, Z. and Iyer, R. K.: CloudVal: A framework for validation of virtualization environment in cloud infrastructure, *Proc. of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)*, pp. 189–196 (2011).
14) Duraes, J. and Madeira, H. S.: Emulation of Software Faults: A Field Data Study and a Practical Approach, *IEEE Transactions on Software Engineering*, Vol. 32, No. 11, pp. 849–867 (2006).
15) Cotroneo, D., Lanzaro, A., Natella, R. and Barbosa, R.: Experimental Analysis of Binary-Level Software Fault Injection in Complex Software, *Proceedings of the IEEE 9th European Dependable Computing Conference (EDCC '12)* (2012).
16) Chou, A., Yang, J., Chelf, B., Hallem, S. and Engler, D.: An Empirical Study of Operating Systems Errors, *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pp. 73–88 (2001).