Characterization of Linux Kernel Behavior under Errors

Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, Zhenyu Yang
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street, Urbana, IL 61801
{wngu, kalbar, iyer, zyang }@crhc.uiuc.edu

Abstract. *This paper describes an experimental study of Linux kernel behavior in the presence of errors that impact the instruction stream of the kernel code. Extensive error injection experiments including over 35,000 errors are conducted targeting the most frequently used functions in the selected kernel subsystems. Three types of faults/errors injection campaigns are conducted: (1) random non-branch instruction, (2) random conditional branch, and (3) valid but incorrect branch. The analysis of the obtained data shows: (i) 95% of the crashes are due to four major causes, namely, unable to handle kernel NULL pointer, unable to handle kernel paging request, invalid opcode, and general protection fault, (ii) less than 10% of the crashes are associated with fault propagation and nearly 40% of crash latencies are within 10 cycles, (iii) errors in the kernel can result in crashes that require reformatting the file system to restore system operation; the process of bringing up the system can take nearly an hour.*

1 Introduction

The dependability of a computing system (and hence of the services provided to the end user) depends to large extent on the error hardiness of the underlying operating system. In this context, analysis of the operating system's failure behavior is essential in determining whether a given computing platform (hardware and software) is able to achieve a desired level of availability/reliability.

The objective of this study is to understand how the Linux kernel responds to transient errors. To this end, a series of fault/error injection experiments is conducted. A single-bit error model is used to emulate error impact on the kernel instruction stream. While the origin of an error is not presumed (i.e., an error can come from anywhere in the system), the injections reflect the ultimate error effect on the executed instructions. This approach allows mimicking a wide range of failure scenarios that impact the operating system[1]. In order to conduct meaningful fault/error injection experiments, it is essential to apply appropriate workloads for generating kernel activity and thus, ensuring a relatively high error activation rate (errors matter to the system only when activated). To achieve this goal, the UnixBench [24] benchmark suite is used to profile kernel behavior and to identify the most frequently used functions representing at least 95% of kernel usage.

[1] Observe that by directly targeting the instruction stream we can emulate not only errors in the code but also errors due to corruption of registers or data. Consider, for example, two scenarios: (i) corruption of the register name in an instruction that uses indirect addressing mode may result in accessing an invalid memory address – equivalent to the register contents corruption; (ii) corruption of an instruction operand that is used as an index to a lookup table containing function offsets may result in accessing an invalid function (address) – equivalent to the look up table data corruption.

Subsequently, over 35,000 faults/errors are injected into the kernel functions within four subsystems: architecture-dependent code (*arch*), virtual file system interface (*fs*), central section of the kernel (*kernel*), and memory management (*mm*). Three types of fault/error injection campaigns are conducted: random non-branch, random conditional branch, and valid but incorrect conditional branch. The data is analyzed to quantify the response of the OS as a whole based on the subsystem and to determine which functions are responsible for error sensitivity. The analysis provides a detailed insight into the OS behavior under faults/errors. The major findings include:

- Most crashes (95%) are due to four major causes: unable to handle kernel NULL pointer, unable to handle kernel paging request, invalid opcode, and general protection fault.
- Nine errors in the kernel result in crashes (most severe crash category*),* which require reformatting the file system. The process of bringing up the system can take nearly an hour.
- Less than 10% of the crashes are associated with fault propagation, and nearly 40% of crash latencies are within 10 cycles. The closer analysis of the propagation patterns indicates that it is feasible to identify strategic locations for embedding additional assertions in the source code of a given subsystem to detect errors and, hence, to prevent error propagation.

2 Related Work

User-level testing by executing API/system calls with erroneous arguments. CMU's Ballista [15] project provides a comprehensive assessment of 15 POSIX-compliant operating systems and libraries as well as Microsoft Win32 API. Ballista bombards a software module with combinations of exceptional and acceptable input values. The responses of the system are classified according to the first three categories of the "C.R.A.S.H" severity scale [16]: (i) *catastrophic* failures (OS corruption or machine crash), (ii) ***r****estart* failures (a task hang), (iii) ***a****bort* failures (abnormal termination of a task).

The University of Wisconsin Fuzz [19] project tests system calls for responses to randomized input streams. The study addresses the reliability of a large collection of UNIX utility programs and X-Window applications, servers, and network services. The *Crashme* benchmark [6] uses *random input* response analysis to test the robustness of an operating environment in terms of exceptional conditions under failures.

Error injection into both kernel and user space. Several studies have directly injected faults into the kernel space and monitored and quantified the responses. FIAT [2] an early

IEEE COMPUTER SOCIETY

fault injection and monitoring environment experiments on SunOS 4.1.2 to study fault/error propagation in the UNIX kernel. FINE [14] injects hardware-induced software errors and software faults into UNIX and traces the execution flow and key variables of the kernel.

Xception [5] uses the advanced debugging and performance monitoring features existing in most of the modern processors to inject faults and to monitor the activation of the faults and their impact on the target system behavior. Xception targets PowerPC and Pentium processors and operating systems ranging from Windows NT to proprietary, real-time kernels (e.g., SMX) and parallel operating systems (e.g., Parix).

MAFALDA [1] analyzes the behavior of Chorus and LynxOS microkernels in the presence of faults. In addition to input parameters corruption, fault injection is also applied on the internal address space of the executive (both code and data segments). In [4], User Mode Linux (equivalent of a virtual machine, representing a kernel) executing on the top of real Linux kernel is used to perform Linux kernel fault injection via the *ptrace* interface.

Other methods to evaluate the operating system. In addition to using fault injection mechanisms, operating systems have been evaluated by studying the source code, collecting memory dumps, and inspecting the error logs. For example, Chou et al. [9] present a study of Linux and OpenBSD kernel errors found by automatic, static, compiler analysis in the source code level. Lee et al. [17] use a collection of memory dump analyses of field software failures in the Tandem GUARDIAN90 operating system to identify the effects of software faults. Xu et al. [26] examine Windows NT cluster reboot logs to measure dependability. Sullivant et al. [23] study MVS operating system failures using 250 randomly sampled reports.

3 Linux Kernel Subsystems

The Linux kernel can be divided into several subsystems [3]. Figure 1, based on [10] shows the size of the code corresponding to each subsystem of the kernel version 2.4.20 released on November 28, 2002.

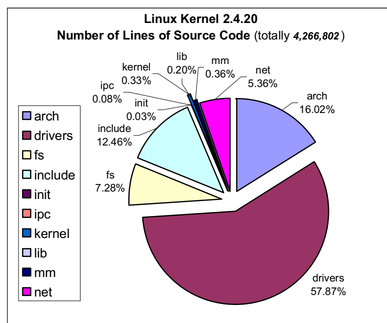


Figure 1: Size of Kernel Subsystems in Terms of Source Code Lines

In our error injection campaigns, we focus on four subsystems: *arch, fs, kernel,* and *mm.* Specifically, (1) *arch* holds the architecture-dependent code (i.e. i386), which includes low-level memory management, interrupt handling, early initialization, and assembly routines, (2) *fs* contains support for various kinds of virtual file systems (we use *ext2* file system), (3) *kernel* is the architecture-independent core kernel code, which includes services such as scheduler, system calls, and signal handling, and (4) *mm* contains high-level architecture-independent memory management code. Selection of the target subsystems is based on the type of activity generated by the benchmark programs (as discussed in Section 4), which for most part invoke functions from the four selected subsystems. Note that the *net* subsystem was not targeted for injection in this study. An important reason was to maintain a single system focus and to keep the study tractable. The network issues can be studied separately.

4 Benchmarks and Kernel Profiling

Due to the size of the kernel, it is impractical to target the entire kernel code for error injection. Depending on the workload, different kernel functions are activated with varying frequency. In order to determine the relative importance of different subsystems and the most frequently used functions, we profile the kernel using the UnixBench benchmark [24]. The use of benchmark programs serves two purposes: (1) it profiles kernel usage to determine targets (most active kernel functions) for error injection campaigns and (2) it creates kernel activity during error injection campaigns to maximize chances for error activation. UnixBench is a UNIX/Linux benchmark suite including tests on CPU, memory management, file I/O, and other kernel components. Eight C programs (*context1.c, dhry, fstime.c, hanoi.c, looper.c, pipe.c, spawn.c* and *syscall.c.)* from the 17 programs included in the benchmark suite are selected for the study. The selection of the programs is to ensure sufficient kernel activity to trigger injected errors and, hence, to enable assessing the kernel behavior in the presence of errors. An additional goal is to ensure that the studied kernel subsystems are thoroughly exercised.

Kernel Profiling. Profiling of the kernel functions while executing the benchmarks is performed using Kernprof (v0.12)[21]. Each activated kernel function is associated with a *profiling value* that indicates the number of times the sampled program counter falls into a given function. A total of 403 kernel functions are profiled. Table 1 gives the distribution of the profiled functions among the kernel modules.

Table 1: Function Distribution Among Kernel Modules

Subsystem Name	Total number of functions within a subsystem	Contribution to the core 32 functions
arch	40	5
fs	154	12
kernel	62	5
mm	71	10
drivers	64	n/a
ipc	1	n/a
lib	6	n/a
net	5	n/a
Total	**403**	**32**

Analysis of profiling data indicates that the top (i.e., most frequently used) 32 functions account for 95% of all profiling values. These functions were selected as the targets for the error injection experiments.

IEEE COMPUTER SOCIETY

5 Experimental Setup and Approach

Failure characterization of the Linux kernel is conducted using software-implemented error injection. Errors are injected to the instruction stream of selected kernel functions. The collected results are analyzed to derive measures characterizing kernel sensitivity to errors impacting the instruction stream.

5.1 Linux Kernel Error Injection Approach

The Linux kernel error injector relies on the CPU's debugging and performance monitoring features and on the Linux Reliability Availability Serviceability (RAS) package [22] to (i) automatically inject errors and (ii) monitor error activation, error propagation, and crash latency.

Linux kernel debugging tools. Linux kernel has several embedded debugging (or failure reporting) tools, including (i) *printk()* – a common way of monitoring variables in the kernel space, (ii) */proc* – a virtual file system for system management (a kernel *executable* core file */proc/kcore* can be debugged by *gdb* to look at kernel variables), (iii) */var/log* – a system log file, and (iv) *Oops message* – provides a kernel memory image at the time of kernel failure.

The above tools, while useful and adequate for most developers, are not sufficient for conducting a comprehensive study characterizing the error sensitivity of the kernel. To enhance error/failure analysis capabilities, we employ the Linux Reliability Availability Serviceability (RAS) package. Specifically, we use SGI's Built-in Kernel Debugger (KDB/KGDB) [20] to enable debugging, including tracing of the kernel code, and the Linux Kernel Crash Dump (LKCD) facility [25] to enable configuring and analyzing of system crash dumps. A set of utilities and kernel patches are created to allow an image of system memory (crash dump) to be captured even if the system abruptly fails. The Linux dump facility LKCD only generates crash dumps under three cases: (i) a kernel *Oops* occurs, (ii) a kernel panic occurs, or (iii) the system administrator initiates a crash dump by typing *Alt-SysRq-c* on the console. To differentiate among reasons for system crashes, custom crash handlers are embedded in the kernel to enable timely invocation of LKCD on crash.

Architecture of the Linux Kernel Error Injector. Mechanisms such as analyzing Oops messages, checking specific log files, and directly using the RAS package, while powerful, are not sufficient when performing large number of kernel error injections. A Linux kernel fault/error injector is designed for such experiments. As shown in the block diagram in Figure 2, the architecture consists of (1) kernel-embedded components – crash handlers, driver, injector, (2) user-level components – injection data producer, injection controller, and data analyzer, and (3) a hardware watchdog to monitor system hangs/crashes and to auto-reboot the kernel in case of failure.

Similarly to Xception[5], the injector uses the debug registers provided by the IA-32 Intel architecture to enable the specifying of the target instruction address and the triggering of the injection. To access the debug registers, an *injection driver* (a kernel module) is developed and attached to the kernel. The controller, in the user space, invokes the injection driver by sending the injection message. The injection driver sets the contents of one of the debug registers to the address of the target instruction. Once the kernel reaches the target address (the program counter matches the contents of the debug register), the *error injector* is activated. The injector carries out the following actions: (1) inserts an error into the binary of the target instruction (i.e., flips a bit), (2) starts a performance counter to measure the latency between the time the corrupted instruction is executed and the actual kernel crash, and (3) returns control to the kernel, which continues from the address of the injected instruction. Figure 3 depicts the process of injecting an error, monitoring the kernel, and recording the crash dump.

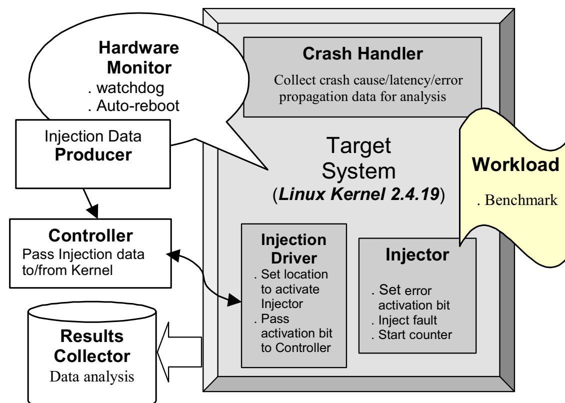


Figure 2: Linux Kernel Error Injector

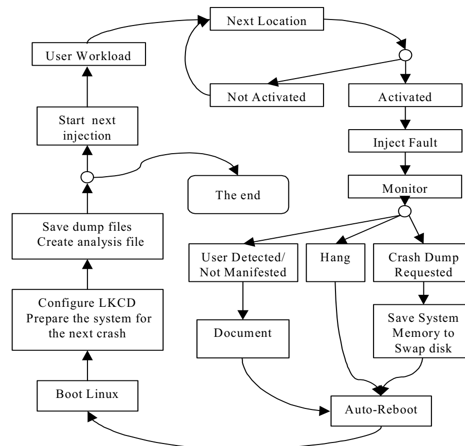


Figure 3: Automated Process of Injecting Errors

An error is injected through the kernel injection module when the target instruction is activated/executed. In case of a crash (1) the system memory is copied into a temporary disk location (a dump device), (2) Linux is booted by crash handler or by watchdog hardware monitor, and the memory image previously saved in the dump device is moved to the dump directory, and (3) the experiments continue with the next error being injected. Observe that the system is rebooted after each run (each single error injection) whenever the target instruction is activated (i.e., the kernel executes the corrupted instruction). If the target instruction is not activated, the experiment proceeds to select the next target instruction without rebooting the system.

IEEE COMPUTER SOCIETY

Crash Handler. The core part of the Linux injector is the crash handler, which invokes the crash dump function of LKCD to save the kernel image at the time of crash. Embedding the crash handler into strategic locations in the kernel enables the collecting of crucial information for discriminating among different categories of crashes and hangs, e.g., kernel panic, divide by zero error, overflow, bounds check, invalid opcode, coprocessor segment overrun, segment not present, stack exception, general protection fault, page fault.

5.2 Error Model

The error model assumed in this study is an error that impacts correct execution of an instruction by the processor. An error can originate in the disk, network, bus, memory, or cache. Single-bit errors are injected to impact the instructions of the target kernel functions. Previous research on microprocessors [7], [8] has shown that most (90-99%) device-level transients can be modeled as logic-level, single-bit errors. Data on operational errors also show that many errors in the field are single-bit errors [13]. Four attributes characterize each error injected:

- *Trigger (when?)* – An error is injected when a target instruction in a given kernel function is reached; the kernel activity is invoked by executing a user-level workload (benchmark) program.
- *Location (where?)* – Error location is pre-selected based on the profiling of kernel functions; the most frequently used kernel functions by the workload are selected for injections. Doing so allows achieving a sufficiently high error activation rate to obtain statistically valid results and conducting the experiments within a reasonable timeframe.
- *Type (what?)* – One single-bit error per byte of an instruction binary is injected.
- *Duration (how long?)* – An injected error persists throughout the execution time of the benchmark program.

5.3 Outcomes, Measures, and Experiment Setup

Outcomes from error injection experiments are classified according to the categories give in Table 3.

Crash latency. The crash latency is defined as the interval between the time an error is injected and the time the error manifests, i.e., the system crashes. To measure the latency, at the end of the error injection routine (part of the error injector), the current value of the performance counter is recorded and subtracted from the value of the counter at the time of error manifestation (recorded by the crash handler routine). Two routines, the *error injector* and the *crash handler*, are used to capture the injection time and the error manifestation time, respectively. Since it takes time for the system to switch between the two routines, simply taking the difference between the two values would include the switching time between routines. Additional measurements were conducted to assess the switching time and subtract it from the calculated crash latency.

Error Propagation. Errors injected and activated in one kernel subsystem may propagate to another subsystem causing the system to crash. Since the kernel is generally divided into several different modules and those modules may interact, it is valuable to analyze error propagation patterns. The injector automatically identifies the Linux kernel subsystem where an error is injected and the subsystem where the crash happens.

Summary of Experiment Setup. Table 2 summarizes the key characteristics of the experimental setup.

Table 2: Experimental Setup Summary

Hardware Platform				Linux OS			Supporting Tools				Error Injection Tool
CPU Type	CPU Clock [GHz]	Cache [KB]	Memory [MB]	Kernel	Distribution	File System	Crash dump	Workload	Profiling	Kernel debug	
Intel P4	1.5	256	256	2.4.19	RedHat 7.3	Ext2	LKCD	UnixBench	Kernprof	KDB	Linux Kernel Injector

Table 3: Outcome Categories

<table>
<tr><th>Outcome Category</th><th colspan="2">Description</th></tr>
<tr><td>Activated</td><td colspan="2">The corrupted instruction is executed.</td></tr>
<tr><td>Not Manifested</td><td colspan="2">The corrupted instruction is executed, however it does not cause a visible abnormal impact on the system.</td></tr>
<tr><td>Fail Silence Violation</td><td colspan="2">Either operating system or application erroneously detects presence of an error or allows incorrect data/response to propagate out.</td></tr>
<tr><td>Crash</td><td>Operating system stops working, e.g., bad trap or system panic.</td><td rowspan="2">Unable to handle kernel NULL pointer dereference, a page fault – the kernel tries to access the bad page pointed by NULL pointer.
Unable to handle kernel page request, a page fault – the kernel tries to access some bad page.
Out of memory, a page fault – kernel runs out of memory.
General protection fault, e.g., exceeding segment limit, writing to a read-only code or data segment, loading a selector with a system descriptor.
Kernel Panic, operating system detected an error.
Trap – invalid opcode, an illegal instruction not defined in the instruction set is executed.
Trap – divide error, a math error.
Trap – init3, a software interrupt triggered by int3 instructions; often used for breakpoint.
Trap – bounds, bounds checking error.
Trap –invalid TSS (task state segment), the selector, code segment, or stack segment is outside the table limit, or stack is not write-able.
Trap – overflow, math error.</td></tr>
<tr><td>Hang</td><td>System resources are exhausted resulting in a non-operational system, e.g., deadlock.</td></tr>
</table>

IEEE COMPUTER SOCIETY

6 Experimental Results

This section presents results from error injection experiments on the selected kernel functions (selected via the profiling discussed in Section 4) while running the benchmark programs. Three types of error injection campaigns are conducted. Table 4 provides brief description of each campaign: ***A*** random injections are made to non-branch instructions, ***B*** conditional branch instructions only are targeted and ***C*** the impact of reversing the logic of conditional branch instruction (i.e., taking a valid but incorrect branch) is studied. Note that, in campaigns ***A*** and ***B*** a target instruction is injected multiple times depending on the number of bytes in the binary representation of the instruction. This section presents the overall outcomes of the three fault injection campaigns followed by a detailed discussion of the error injection outcomes (hang/crash failures, fail silence violations and not manifested errors).

Table 4: Definition of Fault Injection Campaigns

Campaign Name	Target Instructions	Target Bit
A – Any Random Error	All non-branch instructions within the selected function	A random bit in each byte of the instruction
B – Random Branch Error	All conditional branch instructions within the selected function	A random bit in each byte of the branch instruction
C – Valid but Incorrect Branch	All conditional branch instructions within the selected function	The bit that reverses the condition of the branch instruction

6.1 Statistics on Error Activation and Failure Distributions

Figure 4 summarizes the results of the three error injection campaigns. For each campaign, the tables on the left give the statistics on the outcome categories for each targeted kernel subsystem. The number in brackets beside each subsystem indicates the number of functions injected. For example, "arch [6]" indicates that 6 functions from the *arch* subsystem were selected for error injection in a given campaign[2]. For each outcome category, the percentage in the parentheses is calculated with respect to the total number of activated errors. The pie charts on the right provide the overall error distributions for each outcome category [3].

The major findings from over 35,000 fault injections are summarized below.

- Not surprisingly, a significant percentage (35~65%) of injected errors are not activated, i.e. the corrupted instruction is not executed.
- Given that a faulted instruction is executed (i.e. the error is activated), the pie-charts show that for the random branch error, nearly half (47.5%) of the activated errors have no effect (i.e., Not Manifested). This, while at first surprising, is understandable on a close examination of the error scenarios ─ most often the injected error does not change the control path. What is least intuitive is why an error that alters the control path also has no effect. This happens 33% of the time, as indicated in the Figure 4 (Valid but Incorrect Branch). While no single dominant reason can be clearly identified, here for the most part this reflects an inherent redundancy in the kernel code. Representative examples are provided in Section 8.
- Fail silence violations are relatively high for errors in campaign ***A*** (e.g., 6.1% for the *arch* subsystem), and in campaign ***C***, the fail silence violations are the highest (e.g., nearly 18% in the *arch* subsystems). Representative examples for each of these cases are provided in Section 6.5.
- The percentage of Not Manifested Errors in campaign ***B*** is much higher than that of campaigns ***A*** and ***C***. Memory management (*mm*) is the most sensitive subsystem, followed by *kernel* and *fs,* while *arch* is the least sensitive subsystem.
- Although overall the *mm* and *kernel* subsystems are the most sensitive in terms of the percentage of activated errors, in practice three functions, namely *do_page_fault* (page fault handler from *arch* subsystem), *schedule* (process scheduler from *kernel* subsystem), and *zap_page_range* (function from the *mm* subsystem for removing user pages in given range), in random injection cause 70%, 50%, and 30% of crashes in the corresponding subsystems, respectively.
- Nine errors in the kernel result in crashes, which require reformatting the file system. The process of bringing up the system can take nearly an hour.

7 Experimental Results: Crash Analysis

Crash is one of the most severe situations caused by injected errors because it makes the whole system unavailable. In this section, crashes are analyzed from the perspective of their severity, causes, and error propagation.

7.1 Crash Severity

The severity of the crash failures resulting from the injected errors is categorized into three levels according to the system downtime due to the failure. The three identified levels are: (1) *most severe* – rebooting the system after an error injection requires a complete reformatting of the file system on the disk and the process of bringing up the system can take nearly an hour, (2) *severe* – rebooting the system requires the user (interactively) to run *fsck* facility/tool to recover the partially corrupted file system, and although reformatting is not needed, the process can take more than 5 minutes and requires user intervention, and (3) *normal* – at this least-severe level, the system automatically reboots, and the rebooting usually takes less than 4 minutes, depending on the type of machine and the configuration of Linux.

---

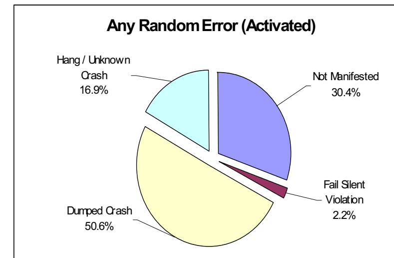
[2] Note that, while all 32 core functions (i.e., contributing to 95% of kernel activity) selected by the kernel profiling are targeted in each error injection campaign, the total number of functions injected in a given campaign is much larger, and different for each campaign. For example, in campaign ***A***, a total of 51 functions are injected (including the top 32 determined by the profiler). This ensures that same core functions are studied in each error injection campaign and that the number of activated errors is sufficient for valid statistical analysis.

[3] In tables in Figure 4, percentages given in the bottom of column *Crash/Hangs* correspond to the sum of (*Dumped Crash + Hang/Unknown Crash*) in the pie-charts.

IEEE COMPUTER SOCIETY

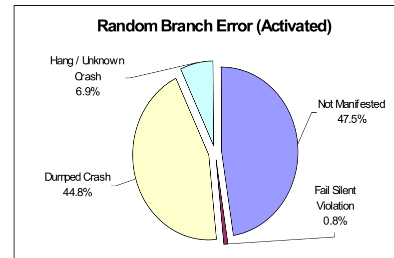
Subsystem [# of Injected Functions]	Injected	Error Activated (Percentage)	Activated		
			Not Manifested	Fail Silence Violation	Crash/Hang
arch[6]	4559	1508(33.1%)	511(33.9%)	92(6.1%)	905(60.0%)
fs[18]	10999	4503(40.9%)	1463(32.5%)	58(1.3%)	2982(66.2%)
kernel[8]	4375	2478(56.6%)	762(30.8%)	0(0.0%)	1716(69.2%)
mm[19]	9044	4881(54.0%)	1330(27.2%)	141(2.9%)	3410(69.9%)
Total[51]	**28977**	**13370 (46.1%)**	**4066(30.4%)**	**291(2.2%)**	**9013(67.4%)**

Any Random Error



Subsystem [# of Injected Functions]	Injected	Error Activated (Percentage)	Activated		
			Not Manifested	Fail Silence Violation	Crash/Hang
arch[10]	428	242(56.5%)	151(62.4%)	6(2.5%)	85(35.1%)
fs[23]	1486	848(57.1%)	419(49.4%)	7(0.8%)	422(49.8%)
kernel[18]	1296	982(75.8%)	442(45.0%)	6(0.6%)	534(54.4%)
mm[30]	1177	727(61.8%)	317(43.6%)	4(0.6%)	406(55.8%)
Total[81]	**4387**	**2779 (63.8%)**	**1329(47.5)**	**23(0.8%)**	**1447(51.7%)**

Random Branch Error



Subsystem [# of Injected Functions]	Injected	Error Activated (Percentage)	Activated		
			Not Manifested	Fail Silence Violation	Crash/Hang
arch[22]	121	58(48.9%)	22(37.9%)	10(17.2%)	26(44.8%)
fs[69]	943	530(56.2%)	200(37.7%)	62(11.7%)	268(50.6%)
kernel[43]	582	317(57.2%)	100(31.5%)	23(7.3%)	194(61.2%)
mm[42]	542	323(59.6%)	87(26.9%)	27(8.4%)	209(64.7%)
Total [176]	**2188**	**1228 (56.1%)**	**409(33.3%)**	**122(9.9%)**	**697(56.8%)**

Valid but Incorrect Branch

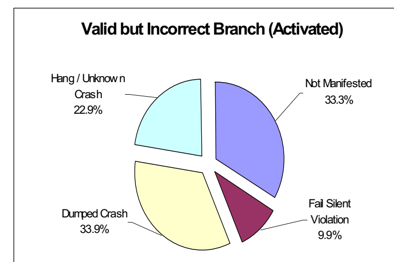


Figure 4: Statistics on Error Activation and Failure Distribution

In all but 34 of 9,600 dumped crashes cases, the system reboots automatically. There are 25 cases in the *severe* level category, and 9 cases require reformatting the file system. Table 5 reports the 9 cases, 4 of which are repeatable and could be traced using *kdb*. A detailed analysis of one of the repeatable crashes (case 9 in Table 5) is provided in Figure 5.

A catastrophic (most severe) error is injected in the function *do_generic_file_read*() from the memory subsystem. The restored (using the *kdb* tool) function calling sequence before the error injection, shown at the bottom right in Figure 5, indicates that *do_generic_file_read()* is invoked by the file system as a read routine for transferring the data from the disk to the page cache in the memory. A single bit error in the *mov* instruction of the *do_generic_file_read*() results in reversing the value assignment performed by the *mov* (see the assembly code at address *0xc0130a33* in Figure 5). As a result, the contents of the *eax* register remain 0x00000080 instead of 0x0000b728, and after executing 12-bit shift (*shrd* instruction in Figure 5), the *eax* is set to *0*.

This is equivalent to corruption of the C-code level variable *end_index* corresponding to the *eax* register; *end_index* is assigned value *0* instead of *0b*. Tracing the C-code shows that another variable (*index)* in *do_generic_file_read()* is initialized to *0* at the beginning of the *for-loop*. However, due to the injected error, the *for-loop* breaks and *do_generic_file_read*() returns prematurely causing subsequent file system corruption; Linux reports: *INIT: ID "1" respowning too fast, 263 Bus error.* Rebooting the system requires reinstallation of the OS.

Additionally, we note that (i) most of the severe crashes happen under campaign ***C***, i.e., reversing the condition of a branch instruction can have a catastrophic impact on the system and (ii) although most often a severe damage to the system usually results in a crash, we observed one case in which the system did not crash after an injected error but could not reboot. The availability impact of the *most severe crashes* is clearly of concern. While a "valid but incorrect branch" error is rare – it is, in our experience, plausible. For example, to achieve 5 nines of availability (5 min/yr downtime) one can only afford one such failure in 12 years, *severe crash* – no more than one in two years, and a *crash* – no more than once a year.

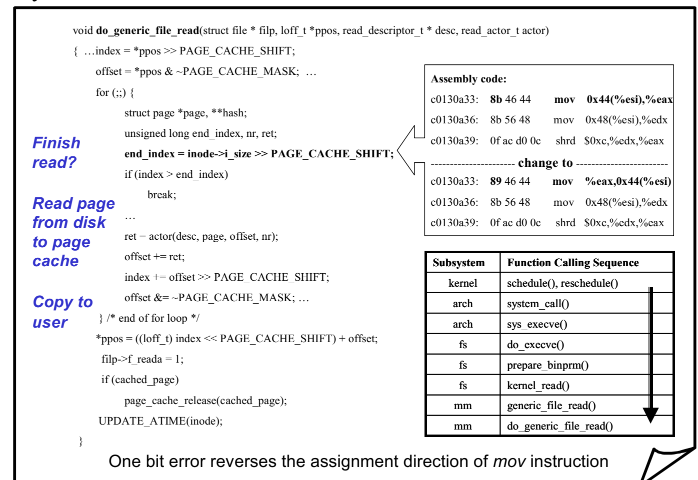


Figure 5: Case Study of a Most Severe Crash

7.2 Crash Causes

The distributions of causes of all dumped crashes are given in Figure 6, where each pie-chart represents an error injection campaign. Major observations are summarized below.

- Regardless of error injection campaign, 95% of the crash causes are due to four major errors: unable to handle kernel null pointer dereference (*null pointer failure*), unable to handle kernel paging request (*paging failure*), *invalid operand/opcode fault,* and *general protection fault*.
- In campaign ***C***, the crash causes are dominated by the invalid operand category (74.7%). Many of those crashes are generated by the assertions inside the Linux kernel. The assertions check the correctness of some specific conditions. At the end of the assertion code, there is a branch instruction. If the check is passed, the branch will follow the normal control flow. Otherwise, it will raise the exception of invalid operand by executing a special instruction of *ud2a*. This is illustrated in the Table 7 (example 4).
- Comparing distributions of crash causes observed in campaigns ***A*** and ***B*** with the distribution obtained from campaign ***C***, one can see the significant difference in the number of paging failures: 35.5%, 36.7% for campaigns ***A*** and ***B***, respectively, versus 3.1% for the campaign ***C***. A detailed case analysis Table 7 (example 2) indicates that paging failures are usually due to random errors leading to corruption of register values. In campaign ***C***, since only one particular bit of a branch instruction is flipped, and therefore, the chance of a paging failure is much smaller.
- The distribution of crash causes in campaign ***A*** is similar to that of campaign ***B***. This phenomenon indicates that, as far as random injections are concerned, the impact of an error in a branch instruction does not differ significantly from the impact of an error in a non-branch instruction.

7.3 Crash Latency

Figure 7 reports the crash latency (in terms of CPU cycles) with respect to target subsystems. The key observations are outlined below.

- The distributions of crash latencies for campaigns ***A*** and ***B*** are similar: 40% of the crashes are within 10 cycles from executing the corrupted instruction.
- In all campaigns, around 20% of crashes have longer latency (>100,000 cycles). This shows that it is fairly common for a crash to happen sometime after an error is injected, indicating the possibility of error propagation (analyzed in the next section).
- For campaign ***C***, the percentage of longer latency errors increases, compared with the other two campaigns. For *fs*, *kernel,* and *mm* subsystems (the cases in *arch* subsystem are statistically insignificant), 40-60% of crash latencies are within 10 cycles. Overall, the crash latencies in this campaign are longer than latencies observed in the other two campaigns. Detailed tracing of crash dumps indicates that random error injections (campaigns ***A*** and ***B***) can corrupt several instructions in a sequence Table 7 (examples 2, and 3). As a result, the system executes an invalid sequence of instructions, which is very likely to cause quick (i.e., short latency) crash Table 7 (example 1). In campaign ***C***, we only reverse the condition of a single branch instruction without affecting any other instructions In this case the system executes incorrect but valid sequence of instructions and thus, a longer latency is observed before the crash.

Table 5: Summary of Most Severe Crashes

No.	Campaign	Repeatability	Injected Subsystem: Function Name	Possible causes for Repeatable Most Severe Crash
1	***C***	Yes	fs: open_nami()	Error results in truncating the file size to 0. No crash is observed, but on reboot, *init* reports: *error while loading shared libraries: /lib/i686/libc.so.6* ***file too short.***
2	***C***	No	mm: do_wp_page()	
3	***C***	No	fs: link_path_walk()	
4	***C***	No	fs: link_path_walk()	
5	***C***	No	fs: sys_read()	
6	***C***	No	fs:get_hash_table()	
7	***C***	Yes	mm: do_wp_page()	Error makes the kernel reuse the page (inside the swap area), which is in use.
8	***C***	Yes	fs: generic_commit_write()	Error reduces the *inode* size *(inode->isize).*
9	***A***	**Yes**	**mm: do_generic_file_read()**	**Undetected error of an incomplete read of the file (data or executable) to the cache page.**

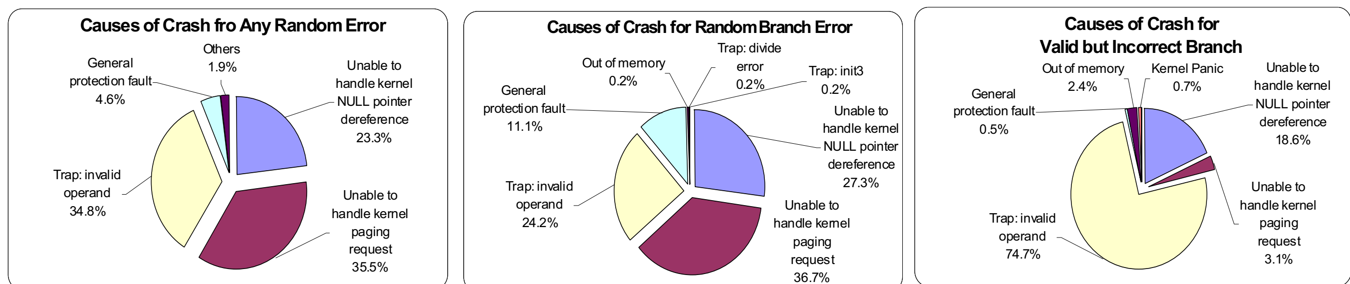


Figure 6: Distribution of Crash Causes

IEEE COMPUTER SOCIETY

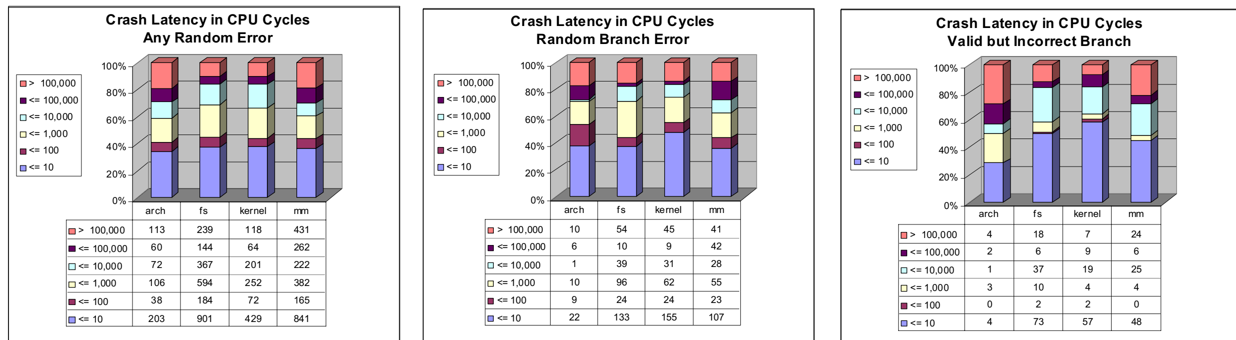


Figure 7: Crash Latency in CPU Cycles

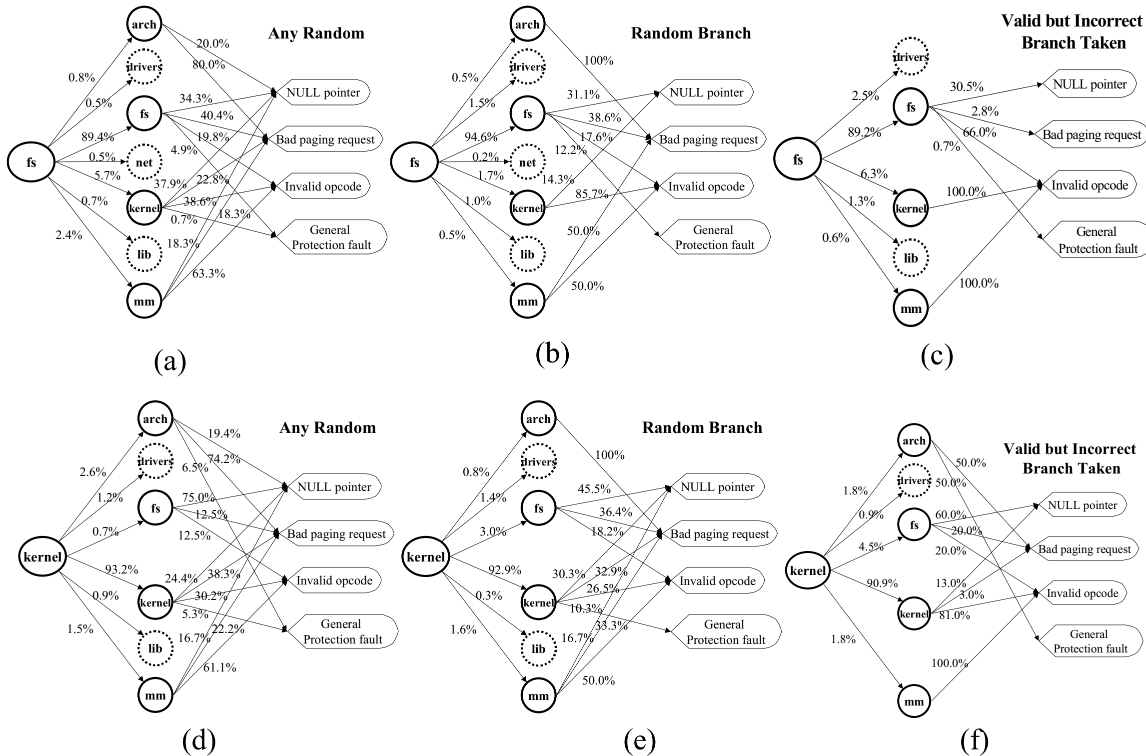


Figure 8: Error Propagation

7.4 Error Propagation

The Linux kernel is a classical monolithic architecture, which means that kernel subsystems are tightly related to each other, even though most of kernel components are accessed via well-defined interfaces. In this section, we study the error propagation between the location of an error injection and that of the system crash. Figure 8 provides error propagation statistics for the two subsystems *fs* (the top three graphs) and *kernel* (the bottom three graphs) for each of the three error injection campaigns[4].

The first node on each graph refers to the faulted subsystem (i.e., the subsystem where an error is injected). The outgoing arcs (transitions) indicate error propagation paths (including the self loop). The end node of each transition corresponds to the subsystem where the crash occurred. The final transition from the end nodes indicates the type of the crash. For example, Figure 8(a) captures crash propagation paths for *fs* subsystem – 89.4% of all crashes happen inside *fs* subsystem, 5.7% of injected errors propagate to and crash in the *kernel* subsystem, and 38.6% of these crashes are due to an invalid operand. Below, we summarize the major findings from the error propagation analysis:

- The overall percentage of error propagation is small (less than 10%). Approximately, 90% of crashes occur inside the subsystem into which the error was injected. This observation agrees with the results from earlier studies on UNIX system behavior, where it has been shown that 8% of errors propagate between the subsystems [14], but it is less than that for the Tandem Guardian operating system, where 18% of software design errors caused error propagation [17].[5]

[4] The analysis of error propagation has been also conducted for the other two target subsystems (i.e., *arch* and *mm*). Due to space limitations, we only report data for the *fs* and *kernel* subsystems.

[5] The study on the impact of transient errors (including error propagation) in LynxOS indicates that from 1% to 4.4% of errors propagate (with the higher percentages observed for application faults) [18].

IEEE COMPUTER SOCIETY

- Errors injected into the *fs* subsystem have the highest probability of propagating. In particular, the primary error (crash) propagation path (5.7%) is to the *kernel* subsystem. Recovery from crash requires reboot the system (around 4 minutes as shown in Section 7.1), which may have a significant negative impact on system availability.
- Several critical error propagation paths can be identified, e.g., from *fs* to *kernel* (graphs (a) to (c)) and from *kernel* to *fs* and from *kernel* to *mm* (graphs (d) to (f) in Figure 8).
- Closer analysis of the propagation patterns indicates that it is feasible to identify strategic locations for embedding additional assertions in the source code of a given subsystem to detect errors and, hence, prevent error propagation. In this scenario, when an assertion fires, an appropriate recovery action (e.g., termination of an offending application process) can be initiated to avoid a kernel crash. Doing so can significantly reduce system downtime and can allow achieving high availability. (Placing of assertion based on error propagation analysis has been also suggested in [11]).

As mentioned earlier, we encounter nine catastrophic kernel crashes, which require reformatting the whole file system. The example analyzed in Section 7.1 shows that an error injected to the *mm* subsystem propagates to the *fs* subsystem and makes the file system unusable. For example, an assertion can be embedded into the kernel code for checking the relationship between the variable *index* and the *inode->i_size* .

8 Experimental Results: Not Manifested Errors and Fail Silence Violations

The pie-charts in Figure 4 show that 30-50% of activated errors do not affect kernel or application functionality (Not Manifested category). A closer case analysis of the examples from this category reveals that the possible causes include redundancy/optimization coding at C source code level and instruction-inherent factors. Examples of such cases are illustrated below.

Redundancy in C Source Code Level. The following piece of code is taken from the function *reschedule_idle()*.

```
212 static void reschedule_idle(struct
task_struct * p)
213 {
214 #ifdef CONFIG_SMP
……
/* shortcut if the woken up task's last
* CPU is idle now. */
best_cpu = p->processor;
223     if (can_schedule(p, best_cpu)) {
```

Valid but Incorrect Branch reverses the direction of the *if* statement in line 223. It turns out that in the single processor machine, *can_schedule* is always true. Consequently without error injection, the body of *if* is taken, which simply reschedules process *p* on the same processor and returns. With an error injected, the body of *if* is not taken, and since there is only one cpu, *p* is still scheduled onto the same processor.

Not Manifested Errors in the Random Branch Error Campaign. Not Manifested errors in campaign ***B*** (Random Branch Error) reach 47% of all activated cases (note that Not Manifested errors in campaigns ***A*** and ***C*** constitute only 33% of all activated errors). This difference can be explained by the intrinsic features of the Linux kernel implementation, namely the fact that in most of the execution scenarios, a given branch is not taken (i.e., the instruction immediately following the branch is executed). The functionality of the branch is virtually the same as a *nop* instruction. Table 6 shows examples of Not Manifested errors observed in campaign ***B***.

Fail Silence Violations in the campaign ***C*** (valid but incorrect branch, in Figure 4) constitute 9.9% of all activated errors. This percentage is substantially higher than those in other two error injection campaigns (2% and 0.8% for campaigns ***A*** and ***B***, respectively). Here, the error-checking scheme of the Linux kernel plays an important role. When the control flow takes a valid but incorrect execution direction, the kernel detects an error and returns with an error code to the user application program. This represents a fail silence violation scenario, since the kernel propagates incorrect data (i.e., notification about an error) to the application.

```
45   /* Seeks are not allowed on
pipes.*/
46      ret = -ESPIPE;
47      read = 0;
48      if (ppos != &filp->f_pos)
49        goto out_nolock;
……
129 out_nolock:
130      if (read)
131        ret = read;
132
133      UPDATE_ATIME(inode);
134          return ret;
135 }
```

The code on the right is taken from the function *pipe_read()*. In line 48, the function performs a check (*ppos != &filp->f_pos*), and if there is an error the control flow moves to line 129, which returns with. an error code (*-ESPIPE*). In campaign ***C***, the condition of the *if* statement at line 48 is reversed. The kernel (falsely) detects an error and returns the error code.

9 Conclusions

This paper describes a series of fault/error injection experiments conducted on the Linux operating system. Using a software-implemented kernel error injector and instrumentation of the Linux kernel, we conduct extensive fault injection campaigns on selected kernel subsystems *arch, fs, kernel* and *mm*. The goal is to analyze and quantify the response of the operating system to a variety of failure scenarios with particular focus on detailed analysis of kernel crashes. Key findings from the experiments are summarized below.

- Most (95%) of the crashes are due to four major causes including unable to handle kernel null pointer dereference, unable to handle a kernel paging request, general protection faults, and invalid operands.
- Nine errors in the kernel resulted in crashes (most severe crash category) which required reformatting the file system. The process of bringing up the system can take nearly an hour.
- Less than 10% of the crashes are associated with fault propagation and nearly 60% of crashes latencies are within 10 cycles.

IEEE COMPUTER SOCIETY

Table 6: Causes of Not Manifested Errors in Random Branch Error Injection Campaign

No	Original Code		After Inject Error		Cause of Not Manifested Errors
	Binary	Assembly	Binary	Assembly	
1	7**4** 56	**je** c01144f4	7**c** 56	**jl** c01144f4	Before error is injected, the status flag is "greater", thus *je* (jump if equal) will not be taken. After error is injected, *jl* (jump if less) is still not taken.
2	0f8**4**ed 000000	**je** c013a9bd	0f8**0**ed 00 00 00	**jo** c013a9bd	Before error is injected, the status flag is "less", thus *je* will not be taken. After error is injected, *jo* (jump if overflow) is still not taken.
3	**7**456	**je c0132548**	**3**4 56	**xor $0x56,%al**	The error injected changes *je* to *xor* which alters the content of register %al. However, the instruction followed is: "mov %ecx, %eax" which assigns correct value to %eax. Thus the error does not manifest.

Table 7: Example Case Studies of Crash Causes

No.	Before error injection (machine code/assembly)	After error injection (machine code/assembly)	Description
1	85 d2 test edx, edx **75** 28 **jne c014c7f1** 31 d2 xor edx, edx … movzbl 0x1b (edx), eax	85 d2 test edx, edx **74** 28 **je c014c7f1** 31 d2 xor edx, edx … movzbl 0x1b(edx), eax	Before error is injected, *edx* is 0x0, *jne* (jump if not equal) is not taken; After error is injected, *je* (jump if equal) is taken; control flow goes to execute *movzbl*, which attempts to access data pointed by NULL pointer (stored in *edx*).
	Unable to handle kernel NULL pointer at 0000001b		
2	8b **51** 0c **mov 0xc**(%ecx),%edx 39 5d 0c cmp %ebx, 0xc(%ebp) 8d 04 82 lea (%edx,%eax,4), %eax 89 45 c0 mov %eax,0xffffffc0 (%ebp)	8b **11** **mov** (%ecx),%edx 0c 39 or $0x39, %al 5d pop %ebp 0c 8d or $0x8d, %al 04 82 add $0x82, %al 89 45 c0 mov %eax, 0xffffffc0(%ebp)	An error makes the original three instructions (*mov, cmp, lea*) to be interpreted as a sequence of five instructions (*mov,or,pop,or, add*). *Pop* modifies *ebp* register, which causes the *mov* instruction to access an incorrect memory location at address ffffffce.
	Unable to handle kernel page request at virtual address ffffffce		
3	**8b** 5d bc **mov** 0xffffffbc(%ebp),%ebx	**cb** **lret** 5d pop %ebp bc in (%dx), %al	Original *mov* instruction is corrupted to *lret*, which causes *general protection fault.*
	General protection fault		
4	74 08 **je** c010510c 0f 0b ud2a	75 08 **jne** c010510c 0f 0b ud2a	C code: *if (!PageLocked(page)) BUG();* Valid but Incorrect Branch error makes control flow go to *BUG*() which is *ud2a* (*invalid opcode* exception).
	invalid opcode		

Acknowledgments

This work was supported in part by a MARCO Program grant SC #1010168/PC#2001-CT-888 Carnegie Mellon and in part by NSF grant CCR 99-02026. We thank Fran Baker for her insightful editing of our manuscript.

References

[1] J. Arlat, et al., "Dependability of COTS Microkernel-Based Systems," *IEEE Transactions on Computers*, 51(2), 2002.

[2] J. Barton, et al., "Fault Injection Experiments Using FIAT, *IEEE Transactions on Computers,* 39(4), 1990.

[3] M. Beck, et al., "Linux Kernel Internals," Second Edition, Addison-Wesley, 1998.

[4] K. Buchacker, V. Sieh, "Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects," *Proc. 3rd Intl. High-Assurance Systems Engineering Symposium,* 2001.

[5] J. Carreira, H. Madeira, and J. Silva, "Xception: A Technique for the Evaluation of Dependability in Modern Computers," *IEEE Transactions on Software Engineering*, 24(2), 1998.

[6] G. Carrette, "CRASHME: Random Input Testing," 1996, http://people.delphiforums.com/gjc/crashme.html

[7] H. Cha, et al., "A Gate-level Simulation Environment for Alpha-Particle-Induced Transient Faults, *IEEE Transactions on Computers*, 45(11), 1996.

[8] G. Choi, R. Iyer and D. Saab, "Fault Behavior Dictionary for Simulation of Device-level Transients," *Proc. IEEE InternationalConf. Computer-Aided Design*, 1993.

[9] A. Chou, et al., "An Empirical Study of Operating Systems Errors," *In Proc. of 18th ACM Symp. on Operating systems principles*, 2001.

[10] M. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study," *Proc. Intl. Conference on Software Maintenance*, 2000.

[11] M. Hiller, et al., "On the Placement of Software Mechanism for Detection of Data Errors," *in DSN-02, 2002.*

[12] M. Hsueh, T. Tsai, and R. Iyer, "Fault Injection Techniques and Tools *IEEE* Computer, 30(4), 1997.

[13] R. Iyer, D. Rossetti, M. Hsueh, "Maesurement and Modeling of Computer Reliability as Affected by System Activity," *ACM Transactions on Computer Systems, Vol.4, No.3, 1986.*

[14] W. Kao, et al, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults," *IEEE Trans. on Software Engineering*, 19(11), 1993.

[15] P. Koopman, J. DeVale, "The Exception Handling Effectiveness of POSIX Operating Systems," IEEE Transactions on Software Engineering, 26(9), 2000.

[16] N. Kropp et al., "Automated Robustness Testing of Off-the-Shelf Software Components," *Proc. FTCS-28, 1998.*

[17] I. Lee and R. Iyer, "Faults, Symptoms, and Software Fault Tolerance in Tandem GUARDIAN90 Operating System," *Proc. FTCS-23,* 1993.

[18] H. Maderia, et al., "Experimental evaluation of a COTS system for space applications," *in DSN-02, 2002.*

[19] B. P. Miller, et al., "A Re-examination of the Reliability of UNIX Utilities and Services," *Tech. Rep.*, University of Wisconsin, 2000.

[20] Built-in Kernel Debugger (KDB), http://oss.sgi.com/projects/kdb/

[21] Kernel Profiling (kernprof), http://oss.sgi.com/projects/kernprof/

[22] Linux RAS Package, http://oss.software.ibm.com/linux/projects/linuxras/

[23] M. Sullivan and R. Chillarege, "Software Defects and Their Impact on System Availability – A Study of Field Failures in Operating Systems," *Proc. FTCS-21,* 1991.

[24] UnixBench, www.tux.org/pub/tux/benchmarks/System/unixbench

[25] D. Wilder, "LKCD Installation and Configuration," 2002. http://lkcd.sourceforge.net/

[26] J. Xu, Z. Kalbarczyk, R. Iyer, "Networked Windows NT System Field Failure Data Analysis," *Proc. of Pacific Rim Intl' Symp. on Dependable Computing*, 1999.