Improving the Reliability of Commodity Operating Systems

Michael M. Swift
mikesw@cs.washington.edu
ACM Member Number: UJ22881

Advisors: Henry M. Levy and Brian N. Bershad
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195 USA

1 Introduction

Despite decades of research in extensible operating system technology, extensions such as device drivers remain a significant cause of system failures. In Windows XP, for example, drivers account for 85% of recently reported failures [17].

This paper describes Nooks, a *reliability subsystem* that seeks to greatly enhance OS reliability by isolating the OS from driver failures. Nooks isolates drivers within lightweight protection domains inside the kernel address space, where hardware and software prevent them from corrupting the kernel. After a fault, Nooks automatically restarts the failed extension.

Our approach to reliability is practical. Rather than guaranteeing complete fault tolerance through a new (and incompatible) OS or driver architecture, Nooks prevents the *vast majority* of driver-caused crashes with *little or no change* to existing driver and system code.

Two factors motivated our research. First, computer system reliability remains a crucial but unsolved problem [10, 14]. While the cost of high-performance computing continues to drop, the cost of failures (e.g., downtime on a stock exchange or e-commerce server, or the manpower required to service a help-desk request in an office environment) continues to rise.

Second, extensions are a leading cause of operating system failure. Extensions are optional components that reside in the kernel address space and typically communicate with the kernel through published interfaces. In addition to device drivers, extensions include file systems, virus detectors, and network protocols. As we mentioned, drivers account for the majority of crashes in Windows XP. In Linux, the frequency of coding errors is seven times higher for device drivers than for the rest of the kernel [5]. While the core operating system kernel reaches high levels of reliability due to longevity and repeated testing, the *extended* operating system cannot be tested completely. With tens of thousands of extensions, operating system vendors cannot even identify them all, let alone test all possible combinations used in the marketplace.

The high cost and pervasive cause of unreliability require systems to become highly tolerant of failures in drivers and other extensions. Furthermore, the millions of existing systems executing thousands of extensions demand a reliability solution that is at once *backward compatible* and *efficient* for common extensions.

To meet this requirement, we implemented Nooks, a prototype reliability subsystem, in the Linux operating system. We experimented with a variety of kernel extension types, including several device drivers, a file system, and a kernel Web server. Using automatic injection of synthetic bugs [11], we show that Nooks can gracefully recover and restart the extension in 99% of the cases that cause Linux to crash. For drivers – the most common extension type – the impact on performance is low to moderate. Finally, of the eight kernel extensions we isolated with Nooks, seven required no code changes, and one required only 13 lines of changes.

2 Related Work

Our work differs from prior work on extensibility and reliability in many dimensions. Nooks supports a conventional programming language, a conventional operating system architecture, and existing extensions. It is transparent to the extensions themselves, supports recovery, and imposes only a modest performance penalty.

Several projects have isolated kernel components through new operating system structures that require the OS and extensions be rewritten. Microkernels [20, 12, 21] move extensions out of the kernel, so that an extension failure does not necessarily crash the system. Transaction-based systems [15, 16] allow the system to recover its state efficiently following an extension fault.

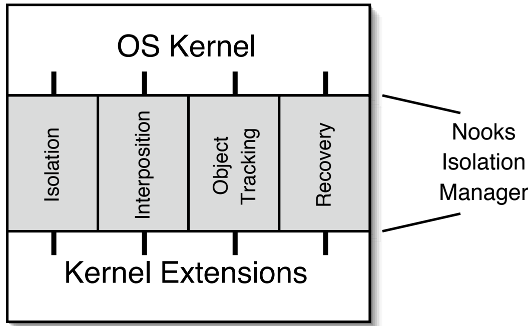

Figure 1: **The Nooks Isolation Manager, a transparent OS layer inserted between the kernel and kernel extensions.**

Type-safe programming languages and run-time systems [3] can prevent many faults types from occurring. However, no commercial OS to date is written in a type-safe language. Hence, this approach does not protect against bugs in existing extensions. In contrast, Nooks supports C, the language used to write most operating systems and extensions today.

Recent years have seen the development of software techniques that enforce code correctness properties, e.g., software fault isolation [19]. These technologies are attractive, but to date have only focused on isolation and do not address recovery. Language-based techniques for verifying the integrity of extensions in existing operating systems have proved effective at revealing programming errors [9, 7, 1, 6]. This static approach obviously complements our own dynamic one.

3 Design

The goals of our design are first, to isolate extensions, so that the kernel is protected from extension faults, and second, to recover extensions after a fault, so that applications depending on the extension can continue to execute. In addition, our design must be backward compatible, so that it applies to existing operating systems and extensions.

We protect the operating system from faulty device drivers by creating a new operating system *reliability layer* that is inserted between the extensions and the OS kernel. The reliability layer mediates all interactions between the extensions and the kernel to facilitate isolation and recovery.

Figure 1 shows this new layer, which we call the *Nooks Isolation manager*. Above the layer is the operating system kernel. The function lines jutting up into the kernel represent kernel-dependent modifications, if any, the OS kernel programmer makes to insert Nooks into a particular OS. These modifications need only be made once. Underneath the layer is the set of isolated extensions. The function lines jutting down represent the changes, if any, the extension writer makes to interface a specific extension or extension class to the system. In general, *no* modifications should be required at this level, since transparency for existing extensions is our major objective.

A crucial property of this layer is *transparency*, i.e. to meet our backward compatibility goals, it must be largely invisible to existing components. The reliability layer provides four major functions transparently to the kernel and extensions, as shown in Figure 1: Isolation, Interposition, Object Tracking, and Recovery. We describe each function below.

3.1 Isolation

The system's *isolation mechanisms* prevent extension errors from damaging the kernel (or other isolated extensions). Every isolated extension executes within its own *lightweight kernel protection domain*. This domain is an execution context with the same processor privilege as the kernel but with limited memory access rights. While the kernel has read-write access to the entire kernel address space, each isolated extension is restricted by virtual memory protection to read-only kernel access and read-write access to its own data. This is similar to the management of address space in some single-address-space operating systems [4]).

To transfer control between protection domains, we created a new kernel service called the *Extension Procedure Call* (XPC) – a control transfer mechanism specifically tailored to isolating extensions within the kernel. This mechanism resembles Lightweight Remote Procedure Call (LRPC) [2] and Protected Procedure Call (PPC) in capability systems [8]. However, LRPC and PPC handle control and data transfer between mutually distrustful peers. XPC occurs between trusted domains but is asymmetric (i.e., the kernel has more rights to the extension's domain than vice versa).

3.2 Interposition

The *interposition mechanisms* transparently integrate existing extensions into the Nooks environment. The interface between the extension, the reliability layer, and the kernel is provided by a set of *wrapper stubs* that are part of the interposition mechanism. The stubs provide transparent control and data transfer between the kernel domain and extension domains. Thus, from the extension's viewpoint, the stubs appear to be the kernel's extension API. From the kernel's point of view, the stubs appear to be the extension's function entry points. The extension loader initiates interposition by linking an extension against wrappers instead of the kernel. As a result, a system administrator may choose, for each extension, whether to use Nooks' isolation services.

3.3 Object Tracking

The *object-tracking functions* oversee all kernel resources used by extensions. In particular, object-tracking code: (1) maintains a list of kernel data structures that are manipulated by an extension, (2) controls all modifications to those structures, and (3) provides object information for cleanup when an extension fails. An extension's protection domain cannot modify kernel data structures directly. Therefore, object-tracking code must copy kernel objects into an extension domain so they can be modified and copy them back after changes have been applied. When possible, object-tracking code verifies the type and accessibility of each parameter that passes between the extension and kernel.

3.4 Recovery

The *recovery functions* detect and recover from a variety of extension faults. The system detects a *software fault* when an extension invokes a kernel service improperly, and detects a *hardware fault* when the processor raises an exception during extension execution, e.g., when an extension attempts to read unmapped memory or to write memory outside of its protection domain.

Recovery consists of two parts. After a fault occurs, the *recovery manager* releases resources in use by the extension. The *user-mode agent* coordinates recovery and determines what course of action to take. The recovery manager is tasked with returning the system, including the extension, to a clean state from which it can continue. Extensions executing in a lightweight kernel protection domain only access domain-local memory directly. All extension access to kernel resources is managed and tracked through wrappers. Therefore, the system can release extension-held kernel structures, such as memory objects or locks, during the recovery process.

By default, the user-mode agent initiates full recovery of faulting extensions by reloading and restarting the extension. It can also change configuration parameters, replace the extension, or even disable recovery if the extension fails too frequently.

4 Reliability Evaluation

The thesis of our work is that we can significantly improve system reliability by isolating the kernel from extension failures. We use automated experiments to demonstrate that Nooks can detect and automatically recover from faults in extensions. In these tests, Nooks recovered from 99% of extension faults that would otherwise crash Linux.

4.1 Test Methodology

We tested our system on a variety of existing kernel extensions and artificially introduced bugs to induce faults.

Extension	Purpose
sb	SoundBlaster 16 driver
es1371	Ensoniq sound driver
e1000	Intel Pro/1000 Gigabit Ethernet driver
pcnet32	AMD PCnet32 10/100 Ethernet driver
3c59x	3COM 3c59x series 10/100 Ethernet driver
3c90x	3COM 3c90x series 10/100 Ethernet driver
VFAT	Win95 compatible file system
kHTTPd	In-kernel Web server

Table 1: **The extensions isolated and the function that each performs. Measurements are reported for extensions shown in bold.**

Our experiments use *synthetic fault injection* [11] to insert faults into Linux kernel extensions. The injector automatically changes single instructions in the extension code to emulate a variety of common programming errors, such as uninitialized local variables, bad parameters, and inverted test conditions.

4.1.1 Types of Extensions Isolated

We used Nooks to isolate three types of extensions: device drivers, a kernel subsystem, and an application-specific kernel extension. The device drivers we chose were common network and sound card drivers, representative of the largest classes of Linux drivers.

The subsystem we chose was the VFAT file system, which is compatible with the Windows 95 FAT32 file system [13]. While drivers tend to have a small number of interfaces with relatively few functions, and hence require few wrappers, the VFAT interface is larger and more complex than the device drivers', and requires far more wrapper.

Lastly, we isolated an application-specific kernel extension – the kHTTPd Web server [18]. kHTTPd resides in the kernel so that it can access kernel network and file system data structures directly, avoiding otherwise expensive system calls. Our experience with kHTTPd demonstrates that our system can isolate even ad-hoc and unanticipated kernel extensions.

Overall, we have isolated eight extensions, as shown in Table 1. Seven of these extensions required no changes to run under Nooks, while the eighth (kHTTPd) required changes to only 13 lines of code. We present reliability and performance results for five of the extensions representing the three extension types: sb, e1000, pcnet32, VFAT and kHTTPd. Results for the remaining three drivers are consistent with those presented.

4.1.2 Test Environment

To measure reliability, we conducted a series of trials in which we injected faults into extensions running under two different Linux configurations. In the first, called

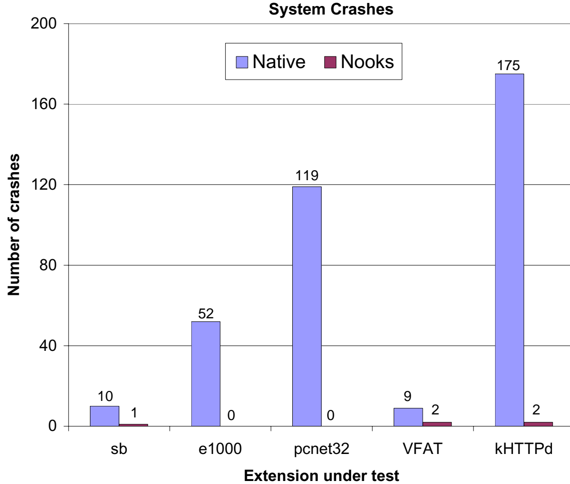

Figure 2: **The reduction in system crashes in 2000 fault-injection trials (400 for each extension) observed using our system. In total, there were 365 system crashes in the native configuration and only five system crashes with Nooks.**

"native," the isolation services were present but unused. In the second, called "Nooks," the isolation services were enabled for the extension under test. For each extension, we ran 400 trials (50 of each fault type) on the native configuration. In each trial, we injected five random errors into the extension and exercised the system, observing the results. We then ran those same 400 trials, each with the same five errors, against Nooks.

4.2 Test Results

Not all fault-injection trials cause faulty behavior, e.g., bugs inserted on a rarely (or never) executed path will rarely (or never) produce an error. However, many trials do cause failures, and we present the results for system crashes.

Figure 2 shows the number of system crashes caused by our fault-injection experiments for each of the extensions running on native Linux and Nooks. Of the 365 crashes observed with native Linux, Nooks eliminated 360, or 99%. In the remaining five crashes the system deadlocked, which our system does not handle. These remaining failures directly reflect our practical approach and are the cost, in terms of reliability, of an approach that imposes reliability on legacy extension and operating systems code. These results show that Nooks is not only able to isolate extensions and prevent the OS from crashing, but also to automatically restart failed extensions. Based on these results, we believe that our system can greatly improve the reliability of existing operating systems.

5 Performance Evaluation

This section presents benchmark results that evaluate the performance cost of the our isolation services. Our experiments use existing benchmarks and tools to compare the performance of a system using Nooks to one that does not. Our test machine is a Dell 1.7 GHz Pentium 4 PC running Linux 2.4.18. The machine includes 890 MB of RAM, a SoundBlaster 16 sound card, an Intel Pro/1000 Gigabit Ethernet adapter, and a single IDE hard disk drive. Our network tests used two similarly equipped machines.

Table 2 summarizes the benchmarks used to evaluate system performance. For each benchmark, we used our system to isolate a *single* extension, indicated in the second column of the table. We ran each benchmark on native Linux without our system and then again on a version of Linux with Nooks enabled. The table shows the relative change in performance for Nooks, either in wall clock time or throughput, depending on the benchmark. We also show CPU utilization measured during benchmark execution, as well as the rate of XPCs per second incurred during each test. The table shows that our system achieves between 44% and 100% of the performance of native Linux for these tests.

As the isolation services are primarily imposed at the point of the XPC, the rate of XPCs offers a telling performance indicator. Thus, the benchmarks fall into three broad categories characterized by the rate of XPCs: low frequency (a few hundred XPCs per second), moderate frequency (a few thousand XPCs per second), and high frequency (tens of thousands of XPCs per second).

We present the results in two groups: device drivers and other kernel extensions. The results are shown in Table 2.

5.1 Device Drivers

We ran three tests on our device drivers. The Play-mp3 benchmark plays an MP3 file through the system's sound card. The Receive-stream test receives a stream of 32KB TCP messages with an isolated Ethernet driver, and the Send-stream test sends a similar stream of messages through an isolated Ethernet driver. Performance was unchanged for the Play-mp3 test, due to its low XPC rate.

Performance for both Receive-stream and Send-stream dropped by only 10%, despite delivering almost 600 Mb/s of data. The Receive-stream test performs only a moderate number of XPCs, while the Send-stream test performs more than 60,000 per second. These XPCs push the Send-stream CPU utilization from 21% on native Linux to 39% with Nooks. In contrast, the CPU utilization for Receive-stream is unchanged. The reason for the additional XPCs is that the Ethernet driver requires

Benchmark	**Extension**	*XPC Rate (per sec)*	*Nooks Relative Performance*	*Native CPU Util. (%)*	*Nooks CPU Util. (%)*
Play-mp3	sb	150	100	4.8	4.6
Receive-stream	e1000 (receiver)	8,923	92	15.2	15.5
Send-stream	e1000 (sender)	60,352	91	21.4	39.3
Compile-local	VFAT	26,979	89	88.7	88.1
Serve-simple-web-page	kHTTPd (server)	61,183	44	96.6	96.8

Table 2: **The relative performance of Nooks compared to native Linux for six benchmark tests. CPU utilization is accurate to only a few percent. Relative performance is determined either by comparing latency (Play-mp3, Compile-local) or throughput (Send-stream, Receive-stream, Serve-simple-web-page).**

a separate XPC for each packet sent. Incoming packets, though, are batched so the driver is able to receive several packets with a single XPC. Despite the higher XPC rate, much of the XPC processing on the sender is overlapped with the actual sending of packets, mitigating some of the Nooks overhead. These results show that our system has only a negligible impact for device drivers with low and moderate XPC rates. When there is spare CPU capacity, our system can have a low performance impact even on drivers with high XPC rates.

5.2 Other Kernel Extensions

Our two other tests are for non-driver extensions. As an indication of application-level file system performance, we measured the time to untar and compile the Linux kernel on a local VFAT file system isolated by Nooks. The final benchmark illustrates the impact of Nooks on the performance of transactional workloads. This benchmark uses kHTTPd on the server, isolated by Nooks, to deliver static content cached in memory.

Table 2 shows that the compilation ran about 10% slower when VFAT was isolated with our system. In this case, the CPU was heavily utilized in the native case, and there was no opportunity to overlap XPCs with other work. Hence, the additional overhead due to Nooks is directly reflected in the end-to-end execution time.

In contrast to VFAT, throughput for kHTTPd dropped almost 60% with Nooks. Two elements of the benchmark's behavior conspire to produce such poor performance. First, the kHTTPd server's processor is the system bottleneck. For example, when run natively, the server's CPU utilization is nearly 96%. Second, kHTTPd requires almost ten XPCs per request. Consequently, the high XPC rate slows the server substantially.

It is clear that kHTTPd represents a poor application of Nooks: it is already a bottleneck and performs many XPCs. This service was cast as an extension so that it could access kernel resources directly, rather than indirectly through the standard system call interface. Since Nooks' isolation facilities impose a penalty on those accesses, performance suffers. We believe that other types of extensions, such as virus and intrusion detectors, which are placed in the kernel to access or protect resources otherwise unavailable from user level, would make better candidates as they do not represent system bottlenecks.

Overall, Nooks provides a substantial reliability improvement at costs that depend on the extensions being isolated. For low and moderate XPC rate extensions, such as sound drivers, Nooks has a negligible cost. For high XPC rate drivers, our system also has a low cost if there is excess CPU capacity. Only for high rate extensions, where there is little CPU capacity to absorb the overhead, does Nooks cost become substantial. The reliability/performance tradeoff is thus one that can be made on a case-by-case basis. For many computing environments, given the performance of modern systems, we believe that the benefits of Nooks' isolation and recovery services are well worth the costs.

6 Conclusions

Kernel extensions are a major source of failure in modern operating systems. Nooks is a new reliability layer intended to significantly reduce extension-related failures. Our system uses software techniques and virtual memory protection to isolate kernel extensions, trapping many common faults and permitting extension recovery. The Nooks system focuses on achieving *backward compatibility*, that is, it sacrifices complete isolation and fault tolerance for compatibility and transparency with existing kernels and extensions. Nevertheless, our work demonstrates that it is possible to realize an extremely high level of operating system reliability with a performance loss ranging from zero to just under 60%. Our fault-injection experiments reveal that Nooks recovered from 99% of the faults that caused native Linux to crash.

Our experience shows that: (1) implementation of a reliability layer is achievable with only modest engineering effort, even on a monolithic operating system like Linux, (2) extensions such as device drivers can be isolated without change to extension code, and (3) isolation and recovery can dramatically improve the system's abil-

ity to survive extension faults. Given the immense number of crashes caused by device drivers and other kernel extensions, we believe that Nooks can have a substantial impact on the reliability of operating systems.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 103–122, May 2001.

[2] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, Feb. 1990.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, Dec. 1995.

[4] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, Nov. 1994.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Lake Louise, Alberta, Oct. 2001.

[6] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN '03 ACM Conference on Programming Language Design and Implementation*, pages 232–244, San Diego, California, USA, June 2003.

[7] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN '01 ACM Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, June 2001.

[8] J. B. Dennis and E. V. Horn. Programming semantics for multiprogramming systems. *Communications of the ACM*, 9(3), Mar. 1966.

[9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, San Diego, CA, Oct. 2000.

[10] A. Gillen, D. Kusnetzky, and S. McLaron. The role of Linux in reducing the cost of enterprise computing, Jan. 2002. IDC white paper.

[11] M. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, Apr. 1997.

[12] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain Resort, Colorado, Dec. 1995.

[13] Microsoft Corporation. FAT: General overview of on-disk format, version 1.03, Dec. 2000.

[14] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kýcýman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report CSD-02-1175, UC Berkeley Computer Science, Mar. 2002.

[15] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 239–253, Pacific Grove, California, Oct. 1991.

[16] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, Oct. 1996.

[17] R. Short. Vice President of Windows Core Technology, Microsoft Corp. Private communication, 2003.

[18] A. van de Ven. kHTTPd: Linux HTTP accelerator, 1999. Available at http://www.fenrus.demon.nl/.

[19] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, North Carolina, Dec. 1993.

[20] W. A. Wulf. Reliatble hardware-software architecture. In *Proceedings of the International Conference on Reliable Software*, pages 122–130, Los Angeles, California, 1975.

[21] M. Young, M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, and A. Tevanian. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93–113, Atlanta, Georgia, June 1986.