Can We Make Operating Systems Reliable and Secure?

Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos

When was the last time your TV set crashed or implored you to download some emergency software update from the Web? After all, unless it is an ancient set, it is just a computer with a CPU, a big monitor, some analog electronics for decoding radio signals, a couple of peculiar I/O devices (e.g., remote control, built in VCR or DVD drive) and a boatload of software in ROM.

This rhetorical question points out a nasty little secret that we in the computer industry do not like to discuss: why are TV sets, DVD recorders, MP3 players, cell phones, and other software-laden electronic devices reliable and secure and computers not? Of course there are many 'reasons' (computers are flexible, users can change the software, the IT industry is immature, etc.), but as we move to an era in which the vast majority of computer users are nontechnical people, increasingly these seem like lame excuses to them. What they expect from a computer is what they expect from a TV set: you buy it, you plug it in, and it works perfectly for the next 10 years. As IT professionals, we need to take up this challenge and make computers as reliable and secure as TV sets.

The worst offender when it comes to reliability and security is the operating system, so we will focus on that. However, before getting into the details, a few words about the relationship between reliability and security are in order. Problems with each of them have the same root cause: bugs in the software. A buffer overrun error can cause a system crash (reliability problem), but it can also allows a cleverly written virus or worm to take over the computer (security problem). Although we focus primarily on reliability below, improving reliability also improves security.

WHY ARE SYSTEMS UNRELIABLE?

Current operating systems have two characteristics which makes them unreliable and insecure: they are huge and they have very poor fault isolation. The Linux kernel has over 2.5 million lines of code and Windows XP is more than twice as large. One study of software reliability showed that code contains 6-16 bugs per 1000 lines of executable code [1] while a different one [7] put the fault density at 2-75 bugs per 1000 lines of executable code, depending on module size. Using a conservative estimate of 6 bugs per 1000 lines of code the Linux kernel probably has something like 15,000 bugs; Windows has as at least double that.

To make matters worse, typically about 70% of the operating system consists of device drivers, and they have error rates 3 to 7 times higher than ordinary code [2], so the above guesses are probably gross underestimates. Clearly, finding and correcting all these bugs is simply not feasible, and bug fixes frequently introduce new bugs.

The large size of current systems means that no one person can understand the whole thing. Clearly, it is difficult to engineer a system well when nobody really understands it. This brings us to the second issue: fault isolation. No single person understands everything about how an aircraft carrier works, either, but the subsystems on an aircraft carrier are well isolated. A problem with a clogged toilet cannot affect the missile-launching subsystem.

Operating systems do not have this kind of isolation between components. A modern operating system contains hundreds or thousands of procedures linked together as a single binary program running in kernel mode. Every single one of the millions of lines of kernel code can overwrite key data structures used by an unrelated component and crash the system in ways difficult to detect. In addition, if a virus or worm manages to infect one kernel procedure, there is no way to keep it from rapidly spreading to others and taking control of the whole machine. Going back to our ship analogy, modern ships have compartments within the hull so if one of them springs a leak, only that one is flooded, not the entire hull. Current operating systems are like ships before compartmentalization was invented: every leak can sink the ship.

Fortunately, the situation is not hopeless. Researchers are busy trying to produce more

reliable operating systems. Below we will address four different approaches that are being taken to make future operating systems more reliable and secure, proceeding from the least radical to the most radical solution.

ARMORED OPERATING SYSTEMS

The first approach, Nooks [8], is the most conservative and is designed to improve the reliability of existing operating systems such as Windows and Linux. It maintains the monolithic kernel structure, with hundreds or thousands of procedures linked together in a single address space in kernel mode, but it focuses on making device drivers (the core of the problem) less dangerous. In particular, it protects the kernel from buggy device drivers by wrapping each driver in a layer of protective software to form a lightweight protection domain as illustrated in Fig. 1. The wrapper around each driver carefully monitors all interactions between the driver and the kernel. The technique can also be used for other extensions to the kernel such as loadable file systems, but for simplicity we will just refer to drivers below.

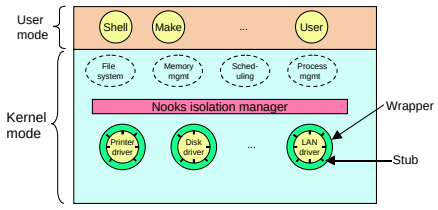

Figure 1. The Nooks model.

The goals of the Nooks project are to (1) protect the kernel against driver failures, (2) recover automatically when a driver fails, and (3) do all of this with as few changes as possible to existing drivers and the kernel. Note that protecting the kernel against malicious drivers is not a goal. The initial implementation was on Linux, but the ideas apply equally well to other legacy kernels.

Isolation

The main tool used to keep faulty drivers from trashing kernel data structures is the virtual memory page map. When a driver runs, all pages outside of it are changed to read-only, thus implementing a separate lightweight protection domain for each driver. In this way, the driver can read kernel data structures it needs but any attempt to directly modify a kernel data structure results in a CPU exception that is caught by the Nooks isolation manager. Access to the driver's private memory, where it stores stacks, a heap, private data structures, and copies of kernel objects is read-write.

Interposition

Each driver class exports a set of functions that the kernel can call. For example, sound drivers might offer a call to write a block of audio samples to the card, another one to adjust the volume, and so on. When the driver is loaded, an array of pointers to the driver's functions is filled in, so the kernel can find each one. In addition, the driver imports a set of functions provided by the kernel, for example, for allocating a data buffer.

Nooks provides wrappers for both the exported and imported functions. When the kernel now calls a driver function or a driver calls a kernel function, the call actually goes to a wrapper that checks the parameters for validity and manages the call as described below. While the wrapper stubs (shown as lines sticking into and out of the drivers in Fig. 1) are generated automatically from their function prototypes, the wrapper bodies must be hand written. In all, 455 wrappers were written, 329 for functions exported by the kernel and 126 for functions exported by device drivers.

When a driver tries to modify a kernel object, its wrapper copies the object into the driver's protection domain (i.e., onto its private read-write pages). The driver then modifies the copy. Upon successful completion of the request, modified kernel objects are copied back to the kernel. In this way, a driver crash or failure during a call always leaves kernel objects in a valid state. Keeping track of imported objects is object specific; code has been hand written to track 43 classes of objects.

Recovery

After a failure, the user-mode recovery agent runs and consults a configuration data base to see what to do. In many cases, releasing the resources held and restarting the driver is enough because most errors are caused by unusual timing conditions (algorithmic bugs are usually found in testing, but timing bugs are not).

This technique can recover the system, but running applications may fail. In additional work [9], the Nooks team added the concept of shadow drivers to allow applications to continue after a driver failure. In short, during normal operation, communication between each driver and the kernel is logged by a shadow driver if it will be needed for recovery. After a driver restart, the shadow driver feeds the newly restarted driver from the log, for example, repeating the IOCTL system calls that set parameters such as audio volume. The kernel is unaware of the process of getting the new driver back into the same state the old one was in. Once this is accomplished, the driver begins processing new requests.

Limitations

While experiments show that Nooks can catch 99% of the fatal driver errors and 55% of the non-fatal ones, it is not perfect. For example, drivers can execute privileged instructions they should not execute; they can write to incorrect I/O ports; and they can get into infinite loops. Furthermore, large numbers of wrappers had to be written manually and may contain faults. Finally, drivers are not prevented from reenabling write access to all of memory. Nevertheless, it is potentially a useful step towards improving the reliability of legacy kernels.

PARAVIRTUAL MACHINES

A second approach has its roots in the concept of a virtual machine, which goes back to the late 1960s [3]. In short, this idea is to run a special control program, called a virtual machine monitor, on the bare hardware instead of an operating system, Its job is to create multiple instances of the true machine. Each instance can run any software the bare machine can. The technique is commonly used to allow two or more operating systems, say Linux and Windows, to run on the same hardware at the same time, with each one thinking it has the entire machine to itself. The use of virtual machines has a well-deserved reputation for extremely good fault isolation—after all, if none of the virtual machines even know about the other ones, problems in one of them cannot spread to other ones.

The research here is to adapt this concept to protection within a single operating system, rather than between different operating systems [5]. Furthermore, because the Pentium is not fully virtualizable, a concession was made to the idea of running an unmodified operating system in the virtual machine. This concession allows modifications to be made to the operating system to make sure it does do anything that cannot be virtualized. To distinguish this technique from true virtualization, this one is called paravirtualization.

Specifically, in the 1990s, a research group at the University of Karlsruhe built a microkernel called L4 [6]. They were able to run a slightly modified version of Linux (L^4Linux [4]) on top of L4 in what might be described as a kind of virtual machine. The researchers later realized that instead of running only one copy of Linux on L4, they could run multiple copies. This insight led to the idea of having one of the virtual Linux machines run the application programs and one or more other ones run the device drivers, as illustrated in Fig. 2.

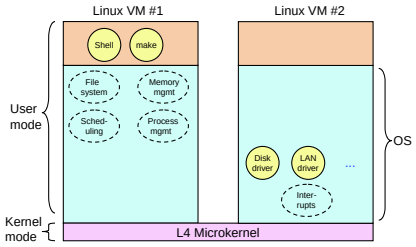

Figure 2. Virtual machines.

By putting the device drivers in one or more virtual machines separated from the main one running the rest of the operating system and the application programs, if a device driver crashes, only its virtual machine goes down, not the main one. An additional advantage of this approach is that the device drivers do not have to be modified as they see a normal Linux kernel environment. Of course, the Linux kernel itself had to be modified to achieve paravirtualization, but this is a one-time change and does not have to repeated for each device driver.

A major issue is how the device drivers actually perform I/O and handle interrupts, since they are running in the hardware's user mode. Physical I/O is handled by the addition of about 3000 lines of code to the Linux kernel on which the drivers run to allow them to use the L4 services for I/O instead of doing it themselves. Furthermore, another 5000 lines of code were added to handle communication between the three drivers ported (disk, network, and PCI bus) and the virtual machine running the application programs.

In principle, this approach should provide a higher reliability than that of a single operating system since when a virtual machine containing one or more drivers crashes, the virtual machine can be rebooted and the drivers returned to their intial state. No attempt is made to return drivers to their previous (pre-crash state) as in Nooks. Thus if an audio driver crashes, it will be restored with the sound level set to the default, rather than to the one it had prior to the crash.

Performance measurements have shown that the overhead of using paravirtualized machines in this fashion is about 3–8%.

MULTISERVER OPERATING SYSTEMS

The first two approaches are focused on patching legacy operating systems. The next two are focused on future ones. The first of these directly attacks the core of the problem: having the entire operating system run as a single gigantic binary program in kernel mode. Instead, only a tiny microkernel runs in kernel mode with the rest of the operating system running as a collection of fully isolated user-mode server and driver

processes. This idea has been around for 20 years, but was never fully explored the first time around because it has slightly lower performance than a monolithic kernel and in the 1980s, performance counted for everything and reliability and security were not on the radar. Of course, at the time, aeronautical engineers did not worry too much about miles per gallon or the ability of cockpit doors to withstand armed attacks. Times change.

Multiserver Architecture

To make the idea of a multiserver operating system clearer, let us look at a modern example, MINIX 3, which is illustrated in Fig. 3. The microkernel handles interrupts, provides the basic mechanisms for process management, implements interprocess communication, and does process scheduling. It also offers a small set of kernel calls to authorized drivers and servers, such as reading a user's address space or writing to authorized I/O ports. The clock driver shares the microkernel's address space, but is scheduled as a separate process. No other drivers run in kernel mode.

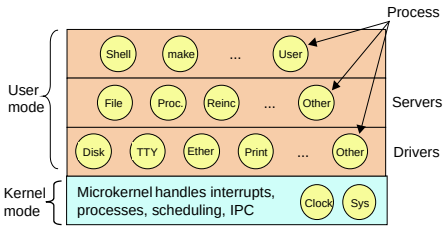

Figure 3. The MINIX 3 architecture

Above the microkernel is the device driver layer. Each I/O device has its own driver that runs as a separate process, in its own private address space, protected by the MMU (Memory Management Unit) hardware. Driver processes are present for the disk, terminal (keyboard and monitor), Ethernet, printer, audio, and so on. The drivers run in user mode and cannot execute privileged instructions or read or write the computer's I/O ports; they must make kernel calls to obtain these services. While introducing a small amount of overhead, this design also greatly enhances reliability, as discussed below.

On top of the device driver layer is the server layer. The file server is a small (4500 lines of executable code) program that accepts requests from user processes for the POSIX system calls relating to files, such as **read**, **write**, **lseek**, and **stat** and carries them out. Also in this layer is the process manager, which handles process and memory management, and carries out POSIX and other system calls such as **fork**, **exec**, and **brk**.

A somewhat unusual server is the reincarnation server, which is the parent process of all the other servers and all the drivers. If a driver or server crashes, exits, or fails to respond to the periodic pings, the reincarnation server kills it (if necessary) and then restarts it from a copy on disk or in RAM. Drivers can be restarted this way, but currently only servers that do not maintain much internal state can be restarted.

Other servers include the network server, which contains a complete TCP/IP stack, the data store (a simple name server used by the other servers), and the information server, which aids debugging.

Finally, above the server layer come the user processes. The only difference with other UNIX systems is that the library procedures for **read**, **write**, and the other system calls do their work by sending messages to servers. Other than this difference (hidden in the system libraries), they are normal user processes that can use the POSIX API.

Interprocess Communication

Interprocess communication (IPC) is of crucial importance in a multiserver operating system, allowing all proceses to cooperate. However, since all servers and drivers in MINIX 3 run as physically isolated processes, they cannot directly call each other's functions or share data structures. Instead, IPC in MINIX 3 is done by passing fixed-length messages using the rendezvous principle: when both the sender and the receiver are ready, the message is copied directly from the sender to the receiver. In addition, an asynchronous event notification mechanism is available. Events that cannot be delivered are marked pending a in bitmap in the process table.

Interrupts are integrated with the message passing system in an elegant way. Interrupt handlers use the notification mechanism to signal I/O completion. This mechanism allows a handler to set a bit in the driver's ''pending interrupts'' bitmap and then continue without blocking. When the driver is ready to receive the interrupt, the kernel turns it into a normal message.

Reliability Features

MINIX 3's reliability comes from multiple sources. To start with, only 4000 lines of code run in the kernel, so with a (conservative) estimate of 6 bugs per 1000 lines, the total number of bugs in the kernel is probably only about 24 (vs. 15,000 for Linux and far more for Windows). Since all device drivers except the clock are user processes, no foreign code ever runs in kernel mode. The small size of the kernel also may make it practical to verify its code, either manually or by formal techniques.

The IPC design of MINIX 3 does not require message queuing or buffering, which eliminates the need for buffer management in the kernel and structurally prevents resource exhaustion. Furthermore, since IPC is a powerful construct, the IPC

capabilities of each server and driver are tightly confined. For each process the available IPC primitives, allowed destinations, and the use event notifications is restricted. User processes, for example, can use only the rendezvous principle and can send to only the POSIX servers.

In addition, all kernel data structures are static. All of these features greatly simplify the code and eliminate kernel bugs associated with buffer overruns, memory leaks, untimely interrupts, untrusted kernel code, and more. Of course, moving most of the operating system to user mode does not eliminate the inevitable bugs in drivers and servers, but it renders them far less powerful. A kernel bug can trash critical data structures, write garbage to the disk, etc.; a bug in most drivers and servers cannot do as much damage since none of these processes are strongly compartmentalized and very much restricted in what they can do.

The user-mode drivers and servers do not run as superuser. They cannot access memory outside their own address spaces except by making kernel calls (which the kernel inspects for validity). Stronger yet, the set of permitted kernel calls, IPC capabilities, and allowed I/O ports are controlled on a per-process basis by bitmaps and ranges within the kernel's process table. For example, the printer driver can be forbidden from *writing* to users address spaces, touching the disk's I/O ports, or sending messages to the audio driver. In traditional monolithic systems, any driver can do anything.

Another reliability feature is the use of separate instruction and data spaces. Should a bug or virus manage to overrun a driver or server buffer and place foreign code in data space, it cannot be executed by jumping to it or having a procedure return to it, since the kernel will not run code unless it is in the process' (read-only) instruction space.

There are other specific features aimed at improving reliability, but probably the most crucial one is the self-healing property. If a driver does a store through an invalid pointer, or gets into an infinite loop, or otherwise misbehaves, it will automatically be replaced by the reincarnation server, often without affecting running processes. While restarting a logically incorrect driver will not remove the bug, in practice many problems are caused subtle timing and similar bugs and restarting the driver will repair the system. In addition, this mechanism allows recovery of failures that are caused by attacks, such as the 'ping of death.'

Performance Considerations

Multiserver architectures based on microkernels have been criticized for decades because of alleged performance problems. However, various projects have proven that modular designs can actually have competitive performance. Despite the fact that MINIX 3 has not been optimized for performance, the system is reasonably fast. The performance loss caused by user-mode drivers is less than 10% and the system is able to build itself, including the kernel, common drivers, and all servers (123 compilations and 11 links) in under 4 sec on a 2.2 GHz Athlon.

The fact that multiserver architectures make it possible to provide a highly-reliable UNIX-like environment at the costs of only a small performance overhead makes this approach practical for wide-scale adoption. MINIX 3 for the Pentium is available for free download under the Berkeley license at *www.minix3.org*. Ports to other architectures and to embedded systems are underway.

LANGUAGE-BASED PROTECTION

The most radical approach comes from a completely unexpected source—Microsoft Research. In effect, it says throw out the whole concept of an operating system as a single program running in kernel mode, plus some collection of user processes running in user mode, and replace it with a system written in new type-safe languages that do not have all the pointer and other problems associated with C and C++. Like the previous two approaches, this one has been around for decades. The Burroughs B5000 computer used this approach. The only language available then was Algol and protection was handled not by an MMU (which the machine did not have) but by the refusal of the Algol compiler to generate 'dangerous' code. Microsoft Research's approach updates this idea for the 21st century.

Overview

This system, called Singularity, is written almost entirely in a new type-safe language called Sing#. This language is based on C#, but augmented with message passing primitives whose semantics are defined by formal, written contracts. Because the system and user processes are all tightly constrained by language safety, all processes can run together in a single virtual address space. This design leads to both safety (because the compiler will not allow a process to touch another process' data) and efficiency (because kernel traps and context switches are eliminated). Furthermore, the Singularity design is flexible since each process is a closed world and thus can have its own code, data structures, memory layout, runtime system, libraries, and garbage collector. The MMU is enabled, but only to map pages rather than to establish a separate protection domain for each process.

A key design principle in Singularity is that dynamic process extensions are forbidden. Among other consequences, loadable modules such as

device drivers and browser plug-ins are not permitted because they would introduce unverified foreign code that could corrupt the mother process. Instead, such extensions must run as separate processes, completely walled off and communicating by the standard interprocess communication mechanism (described below).

The Microkernel

The Singularity operating system consists of a microkernel process and a set of user processes, all typically running in a common virtual address space. The microkernel controls access to hardware, allocates and deallocates memory, creates, destroys, and schedules threads, handles thread synchronization with mutexes, handles interprocess synchronization with channels, and supervises I/O. Each device driver runs as a separate process.

Although most of the microkernel is written in Sing#, a small portion is written in C#, C++, or assembler and must be trusted since it cannot be verified. The trusted code includes the HAL (Hardware Abstraction Layer) and the garbage collector. The HAL hides the low-level hardware from the system by abstracting out concepts such as I/O ports, IRQ lines, DMA channels, and timers to present machine-independent abstractions to the rest of the operating system.

Interprocess Communication

User processes obtain system services by sending strongly typed messages to the microkernel over point-to-point bidirectional channels. In fact, all process-to-process communication uses these channels. Unlike other message-passing systems, which have SEND and RECEIVE functions in some library, Sing# fully supports channels in the language, including formal typing and protocol specifications. To make this point clear, consider this channel specification:

```
contract C1 {
  in message Request(int x) requires x > 0;
  out message Reply(int y);
  out message Error();

  state Start:
   Request? -> Pending;
  state Pending: one {
   Reply! -> Start;
   Error! -> Stopped;
  }
  state Stopped: ;
}
```

This contract declares that the channel accepts three messages, **Request**, **Reply**, and **Error**, the first one with a positive integer as parameter, the second one with any integer as parameter and the third one with no parameters. When used for a channel to a server, the **Request** messages goes from the client to the server and the other two go the other way. A state machine specifies the protocol for the channel.

In the Start state, the client sends the **Request** message, putting the channel into the Pending state. The server can either respond with a **Reply** message or an **Error** message. The **Reply** message transitions the channel back to the Start state, where communication can continue. The **Error** message transitions the channel to the Stopped state ending communication on the channel.

The Heap

If all data, such as file blocks read from disk, had to go over channels, the system would be very slow, so an exception is made to the basic rule that each process' data is completely private and internal to itself. Singularity supports a shared object heap, but at each instant every object on the heap belongs to a single process. However, ownership of an object can be passed over a channel.

As an example of how the heap works, consider I/O. When a disk driver reads in a block, it puts the block on the heap. Later, the handle for the block is passed to the user requesting the data, maintaining the single-owner principle but allowing data to move from disk to user with zero copies.

The File System

Singularity maintains a single hierarchical name space for all services. A root name server handles the top of the tree, but other name servers can be mounted on its nodes. In particular, the file system, which is just a process, is mounted on */fs*, so a name like */fs/users/linda/foo* could be a user's file. Files are implemented as B-trees, with the block numbers as the keys. When a user process asks for a file, the file system commands the disk driver to put the requested blocks on the heap. Ownership is then passed as described above.

Verification

Each system component has metadata describing its dependencies, exports, resources, and behavior. This metadata is used for verification. The system image consists of the microkernel, drivers, and applications needed to run the system, along with their metadata. External verifiers can perform many checks on the image before it is executed, such as making sure that drivers do not have resource conflicts.

Verification is a three-step process:

1. The compiler checks type safety, object ownership, channel protocols, etc.
2. The compiler generates MSIL (Microsoft Intermediate Language), which is a portable JVM-like byte code that can be verified.

3. MSIL is compiled to x86 code by a back-end compiler, which could insert runtime checks into the code (the current back-end compiler does not do this though).

The point of redundant verification is to catch errors in the verifiers.

CONCLUSION

We have examined four different attempts to improve operating system reliability. All of them focus on preventing buggy device drivers from crashing the system. The Nooks approach is to individually hand wrap each driver in a software jacket to carefully control its interactions with the rest of the operating system, but leaves all the drivers in the kernel. The paravirtual machine approach takes this one step further and moves the drivers to one or more paravirtual machines distinct from the main one, taking away even more power from the drivers. Both of these approaches are intended to improve the reliability of existing (legacy) operating systems.

In contrast, the next two approaches are aimed at replacing legacy operating systems with more reliable and secure ones. The multiserver approach runs each driver and operating system component in a separate user process and allows them to communicate using the microkernel's IPC mechanism. Finally, Singularity is the most radical of all, using a type-safe language, a single address space, and formal contracts to carefully limit what each module can do.

What is worth noting is that three of the four research projects use microkernels: L4, MINIX 3, and the Singularity kernel, respectively. It is not known yet which, if any, of these approaches will be widely adopted in the long run. Nevertheless it is interesting to note that microkernels—long discarded as unacceptable due to their lower performance than monolithic kernels—may be making a comeback due to their inherently higher reliability, which many people now regard as more important than performance. The wheel of reincarnation has turned.

ACKNOWLEDGEMENTS

We would like to thank Brian Bershad, Galen Hunt, and Michael Swift for their comments and suggestions. This work was supported in part by the Netherlands Organization for Scientific Research (NWO) under grant 612-060-420. **References**

1. V.R. Basili and B.T. Perricone, ''Software Errors and Complexity: an Empirical Investigation,'' *Commun. of the ACM*, vol. 27, Jan. 1984, pp. 42-52.

2. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, ''An Empirical Study of Operating System Errors,'' *Proc. 18th ACM Symp. on Operating Syst. Prin.*, ACM, pp. 73-88, 2001.

3. R.P. Goldberg, ''Architecture of Virtual Machines,'' *Proc. of the Workshop on Virtual Computer Systems*, ACM, pp. 74-112, 1973.

4. H. Hartig, H. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter, ''The Performance of Microkernel-Based Systems,'' *Proc. 16th ACM Symp. on Operating Syst. Prin.*, pp. 66-77, 1997.

5. J. LeVasseur, V. Uhlig, J. Soess, and Stefan Gotz, ''Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines,'' *Proc. Sixth Symp. on Operating System Design and Impl.* pp. 17-30, 2004.

6. J. Liedtke, ''On Microkernel Construction,'' *Proc. 15th ACM Symp. on Operating Syst. Prin.*, pp. 237-250, 1995.

7. T.J. Ostrand and E.J. Weyuker, ''The distribution of faults in a large industrial software system,'' *Proc. Int'l Symp. on Software Testing and Analysis*, ACM, 2002, pp. 55-64.

8. M. Swift, B. Bershad, and H. Levy, ''Improving the Reliability of Commodity Operating Systems,'' *ACM Trans. on Operating Systems*, vol. 23, pp. 77-110, 2005.

9. M. Swift, M. Annamalai, B. Bershad, and H. Levy, ''Recovering Device Drivers,'' *Proc. Sixth Symp. on Oper. Syst. Design and Impl.*, pp. 1-16, 2004.