The following paper was originally published in the
Proceedings of the USENIX Annual Technical Conference (NO 98)
New Orleans, Louisiana, June 1998

SLIC: An Extensibility System for Commodity Operating Systems

Douglas P. Ghormley, *University of California, Berkeley*
David Petrou, *Carnegie Mellon University*
Steven H. Rodrigues, *Network Appliance, Inc.*
Thomas E. Anderson, *University of Washington*

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

SLIC: An Extensibility System for Commodity Operating Systems

Douglas P. Ghormley
U.C. Berkeley
ghorm@cs.berkeley.edu

David Petrou
Carnegie Mellon University
dpetrou@cs.cmu.edu

Steven H. Rodrigues
Network Appliance, Inc.
steverod@netapp.com

Thomas E. Anderson
University of Washington
tom@cs.washington.edu

Abstract

Modern commodity operating systems are large and complex systems developed over many years by large teams of programmers, containing hundreds of thousands of lines of code. Consequently, it is extremely difficult to add significant new functionality to these systems. In response to this problem, a number of recent research projects have explored novel operating system architectures to support untrusted extensions, including SPIN, VINO, Exokernel, and Fluke. Unfortunately, these architectures require substantial implementation effort and are not generally available in commodity systems.

In contrast, by leveraging the technique of interposition, we have designed and implemented a prototype extension system called SLIC which requires only trivial operating system changes. SLIC efficiently inserts trusted extension code into commodity operating systems, enabling a large class of trusted extensions for existing commodity operating systems such as Solaris and Linux, while retaining full compatibility with existing application binaries. By interposing trusted extensions on existing kernel interfaces, our solution enables extensions which are protected from malicious applications, are enforced upon uncooperative applications, are composable with extensions from other third-party sources, and can be developed at the user-level using state-of-the-art development tools. We have used SLIC to implement and demonstrate a number of useful operating system extensions, including a patch to fix a security hole described in a CERT advisory, a simple encryption file system, and a restricted execution environment for arbitrary untrusted binaries. Performance measurements of the SLIC prototype demonstrate a one-time installation cost of 2-8 μsec and a per-extension invocation overhead commensurate with a procedure call.

This work was supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C-0014), the National Science Foundation (CDA 9401156), Sun Microsystems, California MICRO, Hewlett Packard, Intel, Microsoft, and Mitsubishi. Anderson was also supported by a National Science Foundation Presidential Faculty Fellowship.

1 Introduction

Modifying modern commodity operating systems is extremely difficult and costly. They are large, complex systems developed over many years by large teams of programmers and contain millions of lines of code. It is not unusual for major releases of commodity operating systems to be riddled with flaws introduced during development, typically requiring additional "bug fix" releases which may in turn introduce their own flaws. Compounding these problems, the development and debugging environments for operating system kernels are considerably behind the state of the art. Consequently, it is extremely difficult in practice to add significant new functionality to modern commodity operating systems [12, 1, 34].

Although modifying commodity operating systems is complex and difficult, the need to do so remains. There is a large catalog of operating system functionality which has not been widely deployed, in part because of the difficulty of modifying existing systems: load sharing [51], process migration [43, 12], fast communication primitives [6, 44], upcalls [9], distributed shared memory [25], user-level pagers [49], and novel schedulers [46, 13, 27]. In addition, security flaws are routinely discovered and reported by organizations such as Carnegie-Mellon's Computer Emergency Response Team (CERT) and the Department of Energy's Computer Incident Advisory Capability (CIAC). Despite the need for immediate repair to prevent wide exploitation of these flaws, the required patches can take weeks to become available [42].

This work aims to significantly simplify the process of evolving existing commodity operating systems by enabling new extensions which can manage global resources and/or enforce security guarantees. The ideal system which achieves this goal would possess a number of characteristics: it would require few or no modifications to existing operating systems or applications; it would introduce little overhead; multiple

extensions from independent, third-party sources could be active simultaneously; extensions would be protected from malicious applications and enforced upon uncooperative applications; and kernel extension developers would be able to make use of state-of-the-art user-level development and debugging tools.

Accomplishing this goal would enable independent software vendors (ISV's) to develop and deploy innovative operating system features. In particular, new operating system features developed by research projects could be transferred directly to end users without the need to convince or wait for operating system vendors to adopt the modifications. Furthermore, many CERT and CIAC security advisories normally require the system administrator to wait for a patch from the operating system vendor; instead, the advisory could directly include a small extension to correct the flaw, reducing the window of vulnerability dramatically.

Prior approaches to extending operating systems can be roughly divided into three categories: (i) re-engineering the operating system from the ground up, in the process making it easier to extend, (ii) incrementally re-engineering selected portions of the kernel, and (iii) adding extensions to existing systems without significant modification to either the operating system or its applications.

Over the years, a number of systems have attempted to reduce the cost of adding new operating system functionality by re-engineering the operating system to be extensible. Systems built using this approach include Hydra [48], SPIN [5], VINO [36], Exokernel [14], and Fluke [15]. While many of these systems have successfully demonstrated greatly reduced costs for adding new functionality, the initial cost of replacing existing commodity operating systems is prohibitive; for example, Microsoft spent over $300 M developing Windows NT [50]. Consequently, extensibility architectures developed using this approach will remain unavailable to the average user for the foreseeable future.

A small number of projects have taken the second approach of re-engineering certain kernel interfaces to reduce the complexity of adding new functionality at those interfaces. The `vnode` interface [23] is a prime example of this approach. However, applying this technique to make existing commodity operating systems generally extensible would require modifying and exposing all interfaces where additional functionality is desired, effectively re-engineering the majority of the operating system. Again, for the foreseeable future, such interfaces are unlikely to become generally available in commodity operating systems.

We take the third approach of adding functionality with only minor modifications to the underlying operating system and no modification to application code or binaries. Our work differs from earlier efforts in that our solution—kernel-level interposition of trusted extensions on kernel interfaces—is simple to implement, is efficient, requires no specialized hardware support, protects extensions from malicious or faulty applications, enforces extensions on uncooperative applications, and supports extension stacking. We believe that no other system provides this powerful combination of features for extending existing commodity operating systems.

Prior attempts to extend the operating system without significant modification suffered from significant limitations. Interposition Agents [20] leverages the Mach [1] system call redirection facility to transparently insert user-level extensions at the system call interface. However, because extensions run unprotected in the application's address space and require application cooperation, extensions cannot enforce security guarantees or manage shared resources for competing applications. Software Fault Isolation (SFI) [45] can be used to protect extensions from applications even when loaded in the same address space. Unfortunately, SFI requires a number of compiler optimizations to achieve low overhead and therefore cannot be applied efficiently to existing application binaries. Protected Shared Libraries [4] has the same capability as SFI without the need for compiler optimizations, but does not enforce extensions on applications.

To overcome the limitations of these systems, we have developed SLIC, a prototype system for efficiently inserting trusted extension code into existing operating systems with minor or no modifications to operating system source code. Conceptually, SLIC dynamically "hijacks" various kernel interfaces (such as the system call, signal, or virtual memory paging interfaces) and transparently reroutes events which cross that interface to extensions located either in the kernel (for performance) or at the user-level (for ease of development). Extensions both use and implement the intercepted kernel interface, enabling new functionality to be added to the system while the underlying kernel and existing application binaries remain oblivious to those extensions. SLIC dynamically interposes extensions on kernel interfaces by modifying jump tables or by binary patching kernel routines. The prototype currently runs on Solaris 2.5.1.

We have used the SLIC prototype to implement a number of extensions which would have been significantly more difficult to accomplish by other means. One extension patches a security flaw publicized by CERT [40]. A second extension encrypts file, while a third provides a restricted process execution environment.

The rest of this paper is organized as follows. Sec-

tion 2 provides background on interposition. In section 3, we describe the design, implementation, and performance of SLIC, our prototype interposition system. Three sample extensions and their performance are presented in section 4. In section 5 we discuss our experience with interposition as an extension tool, and the lessons we have learned about building system interfaces to support interposition effectively. Section 6 discusses related work while sections 7 and 8 close with future work and conclusions.

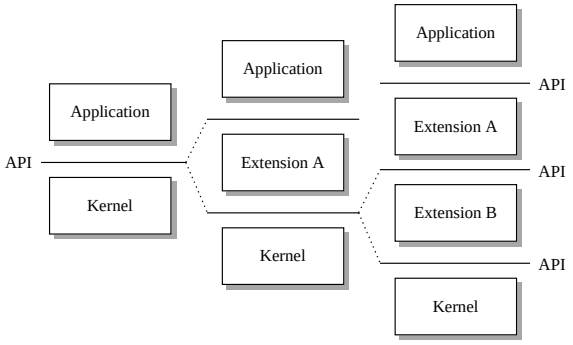

Figure 1: Interposing extensions on an interface. Solid lines indicate interfaces between components. Dotted lines illustrate the interposition of an extension on an interface. Events on the original interface are intercepted and routed through extension code. In this diagram, the original API on the left is maintained at each level on the right, making the interposed extension transparent to the applications, the kernel, and other extensions.

2 Interposition Background

Interposition is the process of capturing events crossing an interface boundary and forwarding those events to an interface *extension*. The extension performs some processing on the event and then either passes the event on to its original destination or forces the event to return. Figure 1 illustrates interposition on an interface by first one extension and then a second.

Interposition has a number of useful properties: it is *transparent*, *incremental*, and *composable*. Interposition is *transparent* because inserted extension code both uses and implements the original interface, enabling user applications and the kernel to remain oblivious to extension code.

Interposition is *incremental* since extensions need only capture the events that they are interested in. Extensions are not required to handle all events crossing the interposed interface boundary, enabling them to leverage the functionality of the existing interface; for example, an extension logging `fork()` system calls can use the underlying operating system's `write()` system call to store the log. Hence, extension writers only have to implement the desired extension functionality, not the functionality of the entire interface.

Because interposition maintains the original interface above and below the extension, it can be applied recursively, enabling multiple, independent extensions to *compose* their functionality. The right-hand side of Figure 1 shows the addition of a second extension to the extension stack. In the diagram, extensions A and B are oblivious of each other's presence, just as the application and kernel are oblivious to the presence of either extension.

The transparency, incrementality, and composability of interposition make it uniquely suitable for extending existing interfaces. Specifically, the transparency of interposition implies that interfaces not designed for extensibility can be extended. Incrementality simplifies extension development by ensuring that extensions need only provide the desired functionality without reimplementing substantial portions of the kernel. Composability means that multiple extensions provided by independent, third-party vendors can be applied to a single interface.

Using kernel-level interposition, extensions have a broad range of capabilities. Extensions can provide security guarantees (e.g., patching security flaws or providing access control lists), virtualize resources (providing cluster-wide process identifiers), modify data (transparently compressing or encrypting files), reroute events (sending events across the network for distributed systems extensions), or inspect events and data (tracing or logging).

However, interposition does have two important limitations. First, interposition requires a well-defined interface on which to capture events. Systems with poorly decomposed functionality may have few such interfaces. Second, new functionality can only be implemented in terms of existing functionality; for example, a cache-coherent file system can only be constructed through interposition if underlying layers expose a cache management mechanism in the file system interface [21].

Despite these limitations, the power and flexibility of interposition has led to its widespread use throughout modern computing systems. Forms of interposition can be found in virtual machines [19], object-oriented programming language systems [22], distributed file systems such as NFS [31], distributed shared memory systems such as TreadMarks [3], the 'pipe' construct of UNIX shells [30], World Wide Web proxy caches [7], MS-DOS terminate-and-stay-resident (TSR) utilities [32] and Macintosh toolbox extensions [11].

3 Design and Implementation

To investigate the suitability of interposition for adding new functionality to existing operating systems, we have designed and implemented SLIC, an interposition system for commodity Unix operating systems. SLIC leverages the transparency, incrementality, and composability characteristics of interposition to provide extensions with the following features:

Security: Extensions are protected from malicious or faulty applications and are enforced on uncooperative applications. This feature enables extensions which manage shared resources and/or enforce security guarantees. (Note that SLIC assumes that extensions are trusted. Other research efforts have addressed issues involved with untrusted extensions [45, 5, 18, 26, 29, 33].)

Ease of Development: During development and testing, extension writers are able to use state-of-the-art programming tools such as symbolic debuggers and performance analysis utilities.

Efficiency: Once development is complete, extensions impose minimal overhead on the system. Per-extension overhead is a few times the cost of a procedure call. Processes that don't use an extension experience a minimal performance slowdown.

3.1 SLIC Architecture

SLIC is comprised of multiple *dispatchers* and *extensions* as well as various *support routines*. Dispatchers are responsible for intercepting system events on a particular interface and for routing those events to interested extensions. Extensions receive events from one or more dispatchers and implement new operating system functionality. Support routines provide extensions with a simple, consistent interface to useful functionality such as memory allocation and synchronization primitives. Each dispatcher may provide additional support routines as appropriate for the interface; for example, our system call dispatcher provides routines to determine the children of a given process.

3.1.1 Dispatchers

Each SLIC dispatcher captures events on a single system interface. Dispatchers use two techniques to intercept interface events. For those interfaces which are invoked via jump tables, such as the system call, `vnode`, and virtual memory interfaces of Solaris, the dispatchers saves the original function address from the jump table and replaces it with the address of its own interception routine. For procedural interfaces which are called directly from various locations in the kernel, such as the signal interface, dispatchers must use binary patching to intercept events. The first few instructions of the relevant procedure are saved and replaced with instructions to jump to the dispatcher whenever the procedure is called. When the original routine needs to be invoked, the saved instructions are executed and control is returned to the original routine at the instruction following the binary patch. Using these techniques, SLIC dispatchers can capture interface invocations at the cost of a procedure call.

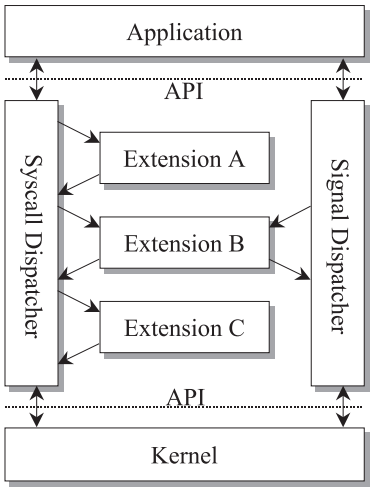

Figure 2: SLIC dispatchers and extensions. The dotted lines represent the interposed interface. Events crossing this interface are captured by a *dispatcher* and forwarded to one or more *extensions*.

Once an event has been captured by a dispatcher, that event is passed to interested extensions for processing. Figure 2 depicts the relationship between dispatchers and extensions. The dispatcher maintains an ordered list of extensions through which intercepted events flow down and return values flow back up. Dispatchers do not make any policy decisions regarding the ordering of extensions—extensions are ordered by the system administrator.

Figure 3 shows a simplified portion of the system call dispatcher interface. Extensions express interest in certain events using a small number of predicates defined by the dispatcher. For example, our system call dispatcher filters system call events based on process identifier and system call type; an extension that only wishes to trace the `open()` system call from process 4191 can specify this easily by invoking `Slic_TraceProc` for process 4191 and calling `Slic_RegisterHandler()` to register a handler only for the `open()` system call. Our signal dispatcher provides similar functionality, enabling filtering on pro-

```
struct Slic_SyscallInfo {
       int syscallNum;
       int args[];
};

struct Slic_ReturnInfo {
       bool forceReturn;
       int  returnValue;
       int  errno;
};

Slic_RegisterExtension(int dispatcherId);
Slic_RegisterHandler(int syscallNum,
        void (handler)(Slic_SyscallInfo *sysInfo,
                       Slic_ReturnInfo *retInfo));
Slic_TraceProc(int pid, bool traceAllChildren);
Slic_UntraceProc(int pid, bool untraceAllChildren);

Slic_IssueSyscall(Slic_SyscallInfo *syscallInfo,
                  Slic_ReturnInfo *returnInfo);
```

Figure 3: A simplified version of the interface exported by the system call dispatcher to extensions.

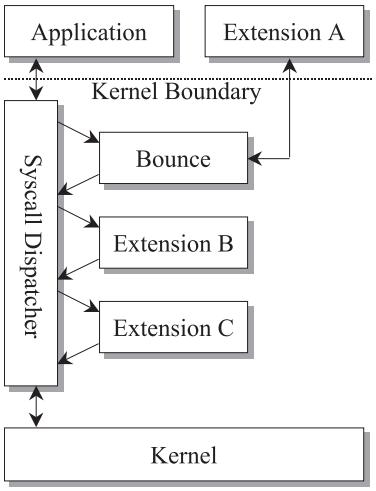

Figure 4: User-level extension environment. Extension A is being developed at the user level while extensions B and C are loaded in the kernel.

cess identifier and signal type.

Upon receiving an event, an extension has a number of options available: the extension can pass the unmodified event down the stack by simply calling `Slic_IssueSyscall()`, the extension can modify the event parameters in `struct Slic_SyscallInfo` and then pass it along, or by setting fields in the `Slic_ReturnInfo` structure, the extension can force the event to return back up the extension stack with an arbitrary return value or error condition (e.g., when a system call would exploit a known security hole). When an event is forced to return, extensions further down the call chain (including the original kernel routines) never see the event. The return value flows back up the chain in reverse order, allowing interested extensions to inspect or modify the returned value. Additionally, while processing one event, extensions may arbitrarily initiate other events on the same interface using `Slic_IssueSyscall()` with a different `Slic_SyscallInfo` structure, capturing the return values as needed. For example, an extension logging `fork()` system calls can initiate a `write()` system call to store the log on disk. Additional events generated by extensions are passed by the dispatcher to the next extension in the chain. To later extensions in the chain, including the kernel, these extension-initiated events are indistinguishable from application-initiated events and are executed with the privileges (and limitations) of the user process.

3.1.2 Extensions

The SLIC architecture enables extensions to be structured in two ways, supporting a tradeoff between ease of extension development and performance. Extensions can be loaded at the user level or in the kernel, as shown in Figure 4, or as a combination of these types. The "Bounce" extension receives events from the dispatcher and hands them off to the user level extension; it also acts as an in-kernel proxy for the extension in order to access the dispatcher's utility functions from the user level. By providing extensions with the same interface whether at the user level or in the kernel, extensions can be easily developed and tested at the user level and then inserted into the kernel for performance. In all cases, the extensions are protected from and enforced upon uncoopertive applications.

Extensions loaded at the user level are each encapsulated in a separate user-level process. This architecture enables extension development to proceed just like standard user applications, with access to user-level libraries (e.g., communication libraries) and state-of-the-art development tools (e.g., symbolic debuggers and performance analysis utilities). User-level extensions are protected from malicious or faulty applications by virtue of running in a separate address space and are enforced on applications by the dispatcher which remains in the kernel. The disadvantage of this approach is that invoking the extension from the dispatcher requires costly context switches and kernel boundary crossings. This organization is similar to that employed by micro-kernels such as Mach [1] and `/proc` based systems such as Ufo [2] but differs in that it supports extension stacking.

To maximize performance once development and testing are complete, extensions can be loaded directly into the kernel where they are invoked directly from

the dispatcher with a procedure call. When events are frequent, this organization has considerably better performance than the user-level approach. In-kernel extensions are protected from malicious applications by virtue of being loaded in the protected kernel region of the address space; they are enforced on uncoopertive applications by the dispatcher. There are two limitations to this approach: (i) the kernel is not protected from malicious or faulty extension code and (ii) there is no support for user-level development tools.

Using a simple upcall/downcall interface, SLIC extensions can also use both models simultaneously. Performance-critical sections of an extension can be located in the kernel, while functionality that is rarely used or which requires access to user-level libraries can be located in a user-level server [37].

3.2 SLIC Implementation

The current implementation of SLIC provides dispatchers on the system call and signal interfaces of Solaris 2.5.1 running on SPARC 10, 20, and UltraSPARC workstations[1]. To minimize kernel modifications, all SLIC components—dispatchers, extensions, and support routines—are dynamically loaded into the kernel as loadable device drivers.

Solaris system calls are routed through the `sysent` table, which contains function pointers to the appropriate system call routines. The system call dispatcher intercepts system call events by replacing entries in this table with pointers to its own dispatch function. Solaris signal delivery proceeds through the `sigaddq()`, `sigaddqa()`, and `sigtoproc()` functions. The signal dispatcher intercepts signals by binary patching these functions at run-time. To enable binary patching, we changed one line of Solaris source code to make the kernel code writable by the SLIC dispatchers, although it should be possible to accomplish this without source code changes by manipulating the memory management hardware directly when SLIC is installed.

The current implementation catches events within the kernel, rather than at the machine level. System calls, for example, are caught at the `sysent` table rather than upon execution of the `trap` instruction. While the two approaches are conceptually similar, intercepting events within the kernel leverages a significant amount of machine-specific code which considerably simplifies the prototype.

[1] Although the SLIC prototype is implemented on Solaris, the principles underlying SLIC are generally applicable; we believe SLIC could easily be ported to other UNIX operating systems.

3.2.1 Shadow Structures

SLIC dispatchers and extensions need a way to record state that persists across event invocations. For instance, the system call dispatcher needs to keep track of which processes are marked for tracing. The appropriate place to store this information is in the process table or the thread structures, but in many operating systems, the size and organization of these structures are compiled into system utilities and other kernel modules.

To minimize modifications to the operating system, we implement *shadow structures* for processes and threads to store state for SLIC dispatchers and extensions. A naïve approach to implementing shadow structures would be to modify process creation and cleanup routines to include a call to manage the shadow structures. Instead, SLIC uses interposition to maintain shadow structures without kernel modifications. Shadow structures are created on demand, when a dispatcher or extension attempts to access a shadow structure. Initialization routines allocate space for the shadow structure and initialize it with a pointer to the underlying kernel's structures. To remove shadow process structures, SLIC interposes on the `wait()` and `waitid()` system calls to detect process death. In Unix, all processes must be waited on, even if by the `init` process. By observing the return values to the `wait()` and `waitid()` system calls, SLIC determines which processes have exited and removes the corresponding shadow structure. Similarly, thread shadow structures can be deleted either on a `thr_exit()` system call or when the thread's process exits.

3.2.2 System Call Buffers

System calls transfer two forms of data: pass-by-value arguments and pass-by-address memory buffers. Inspecting or modifying the pass-by-value arguments in an extension is straightforward. Inspecting or modifying memory buffers located in an application's address space, however, requires more care.

There is a window of vulnerability between the time when an extension inspects or modifies a user-level data buffer and the time when the underlying operating system executes the system call. Other threads in the application can exploit this window of vulnerability to alter the buffer, effectively circumventing any security checks performed by an extension, thus violating the requirement that extensions be enforced on applications. For example, an application may be able to circumvent an access control list extension of the file system by changing the path name of the `open()` system call during this window of vulnerability.

To prevent this situation, before a memory buffer can be inspected or modified by an extension, that buffer

must be protected from modification by the application. Copying these buffers to the kernel provides the necessary protection but introduces a different problem. During system call execution, Unix kernels perform protection checks on memory buffer accesses, requiring that those buffers be located in the application's address space. These checks, performed in the kernel's `copyin()` and `copyout()` routines, are normally necessary to prevent malicious applications from accessing sensitive kernel data. Once a buffer has been copied into the kernel by an extension, the kernel's own security checks will fail when the underlying kernel routines attempt to copy the data, disallowing access to the buffer.

Our solution to this problem is to again apply interposition. SLIC maintains a per-thread list of valid, in-kernel extension buffers and interposes on the `copyin()` and `copyout()` routines in order to permit access to these buffers when appropriate. When an extension allocates an in-kernel buffer to a thread, SLIC adds the address and length of the allocated region to that thread's valid list. When the `copyin()` or `copyout()` routines are invoked, the interposed code checks the target address against the thread's valid list. If the address is a valid extension buffer, then instead of the original `copyin()`/`copyout()` routines, the kernel's `bcopy()` routine is invoked to copy the data to its final location. If the address is not a valid extension buffer, then control is passed to the original copy routine which performs the normal protection checks.

Note that this process introduces an extra copy for data buffers; buffers are copied into the kernel for use by extensions and are later copied again by the underlying kernel routines. Eliminating the extra copy is impractical since numerous kernel routines expect buffers to be copied to stack frames which have not yet been allocated when the extension is invoked.

3.3 Microbenchmarks

To measure the overhead imposed by SLIC on system calls and signals, we ran three microbenchmarks on a 167MHz UltraSPARC running Solaris 2.5.1. Unless otherwise noted, benchmark timings are averaged over 100,000 runs. The first microbenchmark performs a `getpid()` system call, which in Solaris is essentially a null system call. This microbenchmark measures the raw overhead of the system call dispatcher, but does not invoke the modified copy routines since there are no buffers involved. To quantify the overhead resulting from our interposition on the copy routines, the second microbenchmark performs a `sigprocmask()` system call which involves a memory copy of 16 bytes. The `kill()` microbenchmark measures the overhead of the signal dispatcher. It involves a single process sending itself a `SIGUSR1` signal. We use a single process to avoid context switches, thus maximizing the effect of our overhead. Because of the longer run time, results for this benchmark are averaged over 10,000 runs.

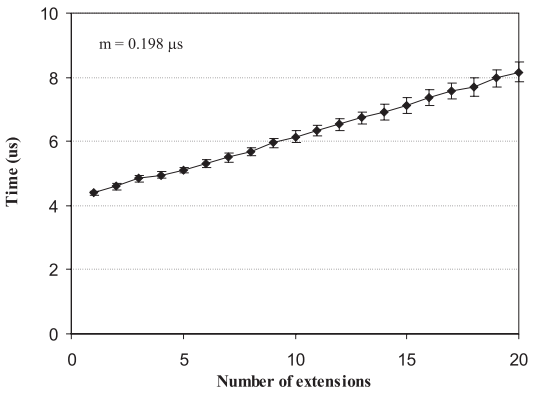

Figure 5: `getpid()` microbenchmark performance with a varying number of extensions. The time on an unmodified system is 2.82 µsec. Error bars indicate one standard deviation above and below the average.

We tested our microbenchmarks with various configurations of SLIC: an unmodified system, a system with SLIC dispatchers but no extensions installed, and a system with the SLIC dispatchers and multiple extensions installed. The results of the `getpid()` benchmark with a varying number of extensions are presented in Figure 5. The incremental cost of adding a new extension is statistically similar for all three benchmarks. Table 1 presents the one-time overhead cost of interposition for each of these benchmarks as well as the time to invoke a user-level extension.

The base overhead of the SLIC dispatchers is approximately 1.5 µsec for system calls and 5 µsec for signals. The incremental cost of loading additional extensions is approximately 0.2 µsec for both dispatchers. The extensions used for these measurements are null extensions which inspect events arguments but not event return values. Inspecting the return values in an extension consumes two additional stack frames and SPARC register windows—one is consumed by the extension while awaiting the return value and a second is consumed by the dispatcher which invoked the extension. Consequently, inspecting return values increases the per-extension cost from 0.2 µsec in Figure 5 to 1.60 µsec. The average cost of a standard procedure call which spills a register window on this platform is approximately 0.8 µsec.

There are two reasons why the overhead for the signal dispatcher is higher than that of the system call dis-

	getpid()		sigprocmask()		kill()	
	Time (µs)	σ	Time (µs)	σ	Time (µs)	σ
Unmodified system	2.82	0.05	3.75	0.20	103.52	1.56
SLIC, no extensions	3.16	0.05	4.19	0.10	108.76	1.53
SLIC, 1 null extension	4.38	0.07	5.56	0.09	109.99	1.49
SLIC, user-level extension	61.83	0.81	60.20	0.88	187.67	2.00

Table 1: Microbenchmark performance of SLIC. For each benchmark, the table shows the average elapsed run time in microseconds and the standard deviation across the runs. The `getpid()` and `sigprocmask()` tests were run with the system call dispatcher loaded while the `kill()` test was run with the signal dispatcher loaded. User-level extension results are averaged over 10,000 runs.

patcher. First, the signal interface is more complicated than the system call interface. Whereas system call events have a calling thread, a system call number, and system call arguments, the signal interface has a calling thread, a signal number, a target process, and an optional target thread. The shadow structure for each process and thread must be determined before extension processing can proceed; there are consequently more shadow structure lookups for the signal interface than the system call interface. The second difference is that intercepting signal events sometimes requires calling thread to pay two interposition costs. Specifically, in Solaris, the `sigaddq()` and `sigaddqa()` routines perform signal queueing side-effects which SLIC extensions may need to leverage; both of these routines call `sigtoproc()` in order to do further signal processing. However, some routines in the kernel invoke `sigtoproc()` directly, requiring SLIC to interpose on that as well to ensure that all signals are seen by the extensions. Those code paths in the kernel which invoke `sigaddq()` or `sigaddqa()` consequently experience the interposition overhead twice.

4 Extensions

To demonstrate the functionality and performance of SLIC, we have implemented prototypes of a variety of extensions: a security patch for a recent CERT advisory, an encryption file system, and a restricted execution environment. Without SLIC, responding to the CERT advisory would have required disabling `admintool` while awaiting a patch, and the encryption file system and restricted execution environment extensions would have required substantial kernel source code modification to achieve the same functionality and performance. Further, without SLIC, adding these features to an existing kernel would require *ad hoc* changes rather than providing a general solution that can be leveraged for future extensions.

4.1 CERT Advisory Extension

The Computer Emergency Response Team (CERT) regularly provides the Internet community with information regarding system security problems. Whenever possible, these advisories include information on how to resolve the reported problem. However, due to the lack of extensibility in existing systems, frequently this advice is to completely disable the insecure feature [39, 40, 41]. However, using SLIC, many of these advisories could be accompanied by small extensions which would resolve the problems without requiring changes to kernel source code. Though operating system vendors do respond to these advisories by supplying patches, those patches can take weeks to become available [42].

To demonstrate patching a security hole in this manner, we have implemented an extension to patch a security hole discovered in the Solaris `admintool` [40] which allowed unprivileged users to truncate arbitrary files. The `admintool` utility creates a local lock file to control access to shared files. By creating a symbolic link at the lock file location, malicious users could cause arbitrary files to be truncated when the lock file was created. Our 100-line extension monitors file operations, preventing symbolic links from being created at the lock location. SLIC thus corrects the security problem while maintaining continued use of `admintool`.

4.2 Encryption File System

In a distributed file system, maintaining file privacy is a primary concern. In a networked environment with a central file server, traditional Unix file protections can be easily circumvented by monitoring network traffic. To protect sensitive files, users may use encryption tools such as PGP [16]. However, stand-alone encryption tools and libraries can be time consuming or cumbersome to use and are not easily integrated with existing applications. A more effective method of ensuring file security is to support file encryption directly in the file system, transparently encrypting file writes

and decrypting file reads when communicating with the server. Rather than rewriting all file systems to support encryption, an easier approach is to implement a single encryption extension which interposes on all file traffic.

We have implemented a simple extension to demonstrate the feasibility of file system encryption using SLIC. This extension implements a trivial *exclusive-or* encryption algorithm similar to that used to test VINO [33]. The prototype extension watches for `open()` and `creat()` system calls of files with a particular suffix and then records the process identifier and the file descriptor returned to the application. On subsequent `read()` or `write()` system calls to these file descriptors, the extension applies a byte-wise `xor` on the data. Key management and encryption algorithms are orthogonal to our demonstration that interposition provides a simple yet powerful means of transforming file system data.

4.3 Restricted Execution Environment

In UNIX, processes run by a user have access to all of the resources granted to that user. There are many cases, however, in which the user does not fully trust the program being run. For example, programs downloaded from untrusted sources may actually be Trojan horses designed to steal or destroy information [47, 10]. In addition, there are cases in which the user trusts the program, but not the data being processed, as in the case of web browser helper applications used by web browsers to display various data formats. Input data could potentially exploit bugs in helper applications such as `ghostview` to insert viruses into the system [38].

The tracing facility of the standard Solaris `/proc` file system is one method that has been used to construct a restricted execution environment [17]. Potentially insecure system calls are captured and then selectively denied or altered. The `/proc` approach, however, suffers from two primary shortcomings. First, intercepting system calls using `/proc` is expensive, requiring two context switches over the base system call overhead. This is especially problematic for system-call intensive applications. Second, systems using `/proc` cannot properly handle the system call buffer problem described in section 3.2.2. While `/proc` does enable an extension to inspect system call buffer data, a multi-threaded application may maliciously modify the buffer between extension validation of the buffer and kernel use of the buffer, effectively subverting the security system. Using SLIC we have implemented a restricted execution environment extension that does not have these limitations.

This extension is a modified version of the Janus system [17] and provides the user with a configurable security environment. For example, applications can be given a subset of read/write/execute access to any number of directory subtrees. The ability to `fork()` or to perform a variety of other system calls can be disabled. Any attempts by the application to perform a restricted operation results in the extension returning an `EPERM` error. The extension monitors only the subset of system calls necessary to maintain the security guarantees, thus minimizing overhead. When traced applications invoke restricted system calls, the extension checks the arguments to the call and determines if the call should be allowed or denied. The restricted environment used for benchmarking denies 45 system calls outright (e.g., `chown()`) and performs security checks, such as checking the path or file access permissions, for 21 additional system calls (e.g., `rmdir()`).

4.4 Performance

To evaluate the impact of these extensions on system performance, we ran the extensions under three benchmarks: the Modified Andrew Benchmark [28], a TEX compilation of a 494-page (1.32 MB) document, and a `gcc` compilation of `emacs-19.34` without support for X Windows. The Modified Andrew Benchmark consists of multiple phases which create directory subtrees, copy files, search file attributes via `find`, search files for a text string via `grep`, and compile files. While this benchmark fits entirely in the file cache of modern systems and is therefore no longer useful for measuring file system performance, this benchmark is useful for exposing the overhead imposed by SLIC. The TEX and `gcc` benchmarks were chosen to be representative of document processing and compilation workloads. All measurements were run on a 167MHz UltraSPARC running Solaris 2.5.1 with all benchmark data files placed in a memory-mounted `/tmp` file system. Table 2 reports some relevant statistics for each benchmark.

Table 3 presents the results of running the benchmarks with each extension as well as with all extensions simultaneously. Since the CERT extension does not catch any system calls issued by the benchmarks, as indicated by Table 2, it effectively acts as a null extension for these tests; consequently, the overhead depicted in Table 3 for the benchmarks running on CERT is exclusively due to the SLIC dispatchers and infrastructure. There are three anomalies in the table worthy of note: the MAB and `gcc` benchmarks appears to run faster with all three extensions loaded than with just the Encrypt extension loaded and the `gcc` benchmark appears to run faster with SLIC loaded than on the baseline system. However, an inspection of the standard de-

	Procs	Total System Calls	System Calls Caught					
			CERT		Encrypt		REE	
MAB	471	40500	0	0%	2732	7%	9246	23%
TEX	3	2748	0	0%	1635	60%	703	25%
gcc	379	140656	0	0%	11031	8%	47888	34%

Table 2: Benchmark characterization. For each benchmark, this table presents the total number of processes created during a run, the total number of system calls issued by those processes, and the number and percent of those system calls which are caught by each extension. "MAB" is the Modified Andrew Benchmark and "REE" is the Restricted Execution Environment.

	MAB			TEX			gcc		
	Time (*s*)	σ	S	Time (*s*)	σ	S	Time (*s*)	σ	S
Baseline	15.71	0.10		14.50	0.30		160.30	1.99	
SLIC, no extensions	15.92	0.16	1%	15.06	0.23	4%	159.13	0.34	-1%
CERT	16.12	0.30	3%	15.09	0.39	4%	160.11	0.55	0%
Encrypt	17.69	0.85	13%	15.87	0.53	9%	168.30	0.53	5%
REE	15.93	0.33	1%	15.15	0.73	4%	160.89	1.63	0%
CERT + Encrypt + REE	17.24	0.09	10%	15.90	0.37	10%	166.98	3.43	4%

Table 3: Benchmark performance on sample extensions. "Baseline" represents a machine without any SLIC dispatchers or extensions loaded. "SLIC, no extensions" represents dispatchers loaded, but no extensions; this row measures the effect of the SLIC interposition overhead. The rows labeled "CERT", "Encrypt" and "REE" present the benchmark elapsed times with a single extension loaded. The last line shows benchmark performance with all three extensions interposing simultaneously. For each benchmark, the table shows the average elapsed run time in seconds, the standard deviation across the runs, and the percent slowdown ("S").

viation in each of these cases reveals that the anomalies are well within one standard deviation of the mean and are likely to be experimental variance. Though SLIC imposes a certain amount of overhead on applications, the last line in Table 3 illustrates that much of the overhead experienced by the benchmarks is due to the SLIC dispatcher infrastructure, a cost which is only paid once; the per-extension overhead is small for these workloads.

5 Interposition Evaluation

This section describes our experiences with implementing SLIC and presents a number of general principles for developing interfaces that are conducive to interposition. Although SLIC was designed for and will work with existing operating systems, there are a number of improvements that can be made to make these systems more interposition-friendly. We have drawn these lessons from our implementations of the system call and signal dispatchers for Solaris 2.5.1, as well as preliminary analyses of the virtual memory mechanism of Solaris, the process scheduler interface of FreeBSD 2.2.5R, and the system call interface of Linux 2.0.

The problems that we have encountered can be divided into four categories: the asymmetric trust mechanisms of the system call interface, the implicit event information in the interfaces we interposed on, an incomplete decomposition of functionality in the system, and miscellaneous implementation issues.

5.1 Asymmetric Trust Mechanisms

The solution adopted by the current prototype to correctly handle system call buffers (see section 3.2.2) introduces a second copy for those data buffers which must be securely examined by an extension. This second copy is not fundamental to interposition but rather arises out of idiosyncrasies of the Solaris system call routines. In Solaris, as in most Unix systems, system call buffers must be copied into the kernel before they can be used. The copies are only necessary on interfaces such as the system call interface which have an asymmetry of trust—applications trust the kernel, but the kernel does not trust applications. Extensions which interpose on such interfaces are viewed by the kernel as being part of the untrusted user application. Interfaces which have a symmetry of trust do not need to copy data buffers before using them because it is assumed that no other threads in the trusted domain will maliciously modify the buffers to subvert the system.

Consequently, the ideal interface for interposition is one with a symmetry of trust. For the system call in-

terface, this would mean interposing on events *after* the data buffer addresses had been validated and the buffers themselves had been copied into the kernel. In Solaris, this is impractical because the copy routine invocations are scattered throughout the system call handler routines. The ideal interposition-friendly interface would perform any required security checks and copies in separate routines before invoking system call handlers. This approach would eliminate the unnecessary second copies and would remove the need for SLIC to interpose on the kernel's copy routines.

5.2 Implicit Event Information

Many interfaces in today's operating systems rely on *implicit information*—information that is not passed directly as an argument to the event but rather is stored in global data structures. For example, when a system call is made, the process identifier of the application is not passed to the system call handler. Similarly, the return value of a system call is not returned explicitly according to normal calling conventions, but is stored in the kernel's process structure. The credentials determining a process's right to open a file or send a signal are also stored in the process structure. Accessing this implicit information requires an understanding of complicated kernel structures and kernel locking conventions.

To minimize the dependency of extensions on a particular version of the operating system, SLIC dispatchers provide extensions with simple utility functions to access a variety of implicit event information, hiding the details of kernel data structures and locking conventions. Unfortunately, providing these functions in the dispatchers increases the dependency of those dispatchers on the particular version of the operating system. An interposition-friendly interface would make implicit event information readily accessible when an event is raised, enabling SLIC to be more easily ported to new operating system versions.

Implicit event information also limits the functionality of extensions, as in the case of implicit credentials. If extensions cannot modify event credentials, then extensions would be restricted to the rights of the calling thread. For example, an extension could not write to files owned by `root` unless invoked from a `root` process. A naïve approach to circumvent this problem would be to enable extensions to modify a process's credentials stored in global data structures for the duration of the intercepted event. However, in a multithreading environment, other threads running concurrently may access the modified credentials, producing unpredictable results. Including credentials as explicit parameters to the event enables extensions to modify the credentials as necessary.

The `ioctl()` system call poses a different problem for interposition. Originally designed as a way to manage arbitrary devices, `ioctl()` system calls have a variable number of parameters, nearly any of which may reference arbitrary memory buffers which may in turn contain pointers to other memory buffers. The meaning of the arguments and the structure of each buffer is defined by the particular device driver. Hence an extension cannot know in advance how to handle the arguments of any given `ioctl()` call. This is problematic for extensions such as system call logging or security extensions which may need to understand the arguments of an `ioctl()`. While it is possible to derive some information about device/application interactions by interposing on the `copyin()`/`copyout()` routines, general interpretation of `ioctl()` semantics is difficult. Events with run-time determined semantics are not conducive to interposition.

5.3 Separation of Policy and Mechanism

To reduce the dependencies on a particular hardware platform or operating system version, extensions should interpose on system policies while leveraging system mechanisms. A clean separation of policy and mechanism in the compiled kernel is therefore critical. However, this principle of decomposition is violated by today's operating systems. While experimenting with extending the scheduler interface in FreeBSD 2.2.5R, we found that a single scheduler routine, `cpu_switch()`, implements both the policy of selecting the next process to run as well as the actual context switch mechanism. Enabling interposition on the policy while leveraging the existing mechanism required separating the policy and mechanism into two routines, creating a procedural interface which could be interposed on.

A variation of this lack of separation occurs with the Linux routines for accessing system call data buffers, `getuser()` and `putuser()`. These routines are inlined at compile time, making interposition at run time extremely difficult. Interposing on the copy routines requires modifying the Linux source to disable inlining of those copy routines, again creating a procedural interface that can be interposed on, albeit at a small cost in performance.

5.4 Miscellaneous Issues

Our method of using binary patching to intercept procedural invocations is simple to implement, but requires that kernel code be writable. Unfortunately, Solaris is loaded such that the kernel code is read-only,

preventing binary patching. By modifying a single line of Solaris source, we were able to make the kernel code writable.

Managing shadow structures for processes and threads complicates SLIC and increases kernel memory consumption. Adding a hook to the kernel's process and thread structures would eliminate these problems.

6 Related Work

There has been a considerable amount of recent work [36, 5, 14, 15] that has explored novel kernel designs for extensible operating systems. Of these systems, SPIN [5] and VINO [36] are the closest in concept to our work. Both offer extensibility through interposition on a number of kernel interfaces, but have explicitly crafted those interfaces for extensibility. SPIN and VINO also aggressively focus on ensuring kernel protection from untrusted extensions, SPIN by using a type-safe language [35, 18], and VINO through software fault isolation [45] and in-kernel transactions [33]. In contrast, SLIC assumes trusted extensions and focuses on an evaluation of the technique of interposition and its suitability for legacy operating systems.

Interposition Agents [20] demonstrated the usefulness of constructing extensions in terms of the high-level abstractions of an interface (such as path names), rather than the low-level events crossing that interface (such as `open()`). Interposition Agents used the system call redirection facility of Mach which bounces system calls to extensions linked into an application's address space. Consequently, extensions are neither protected from nor enforced on applications and thus cannot implement security extensions or share data between distrustful applications. Additionally, the multiple protection boundary crossings limit the performance of the system. SLIC enables high-performance interposition that is both enforced on and protected from applications, enabling a significantly larger class of extensions. In principle, the toolkit presented in [20] could be ported to SLIC, further simplifying the process of extension development.

COLA [24] enables interposition at the system call interface, but without any modification of the operating system kernel. It operates through interposition at the library level and consequently suffers from the same security drawbacks as the Mach interposition technology described above.

Protected Shared Libraries (PSL) [4] enables extensions to be securely loaded into an application's address space, so that user programs cannot access or modify extension code or data. PSL does not provide a mechanism for enforcing extensions on applications, which SLIC does. PSL is primarily intended for adding new interfaces to a system, although it could be combined with the interposition mechanisms used in SLIC to enable modification of existing interfaces. Finally, the PSL protection technique relies on per-thread segment protections supported by the IBM RS/6000 architecture, while the principles in SLIC are generally applicable across a variety of operating system platforms.

Disco [8] and Fluke [15] are virtual machine monitors which use strategies similar to those of SLIC for different purposes. Disco uses interposition and binary rewriting to ease the implementation of operating systems for new architectures, rather than adding extensibility to existing operating systems, as SLIC does. Fluke uses interposition for extensibility, relying on a heavy decomposition of services into nested process domains, instead of adding extensibility to an existing kernel.

7 Future Work

While interposition enables extension stacking, the mechanism cannot guarantee that the extensions will be compatible with each other. Some extension combinations may only impair performance (e.g., for many types of data, it is more efficient to execute a data compression extension before a data encryption extension) while others may impair correctness (e.g., an extension which needs to inspect the magic number at the head of executables may run incorrectly when invoked after an encryption extension). Prior experiences with conflicts among MS-DOS terminate-and-stay-resident (TSR) utilities and Macintosh toolkit extensions indicate that a method for identifying and managing conflicts among incompatible extensions is sorely needed.

8 Conclusions

This paper has examined the utility of interposition as a mechanism for making commodity operating systems extensible. We have shown that interposition is suitable to a number of useful extensions, and we have presented a prototype system, SLIC, which enables operating system extensibility through interposition in Solaris with minimal kernel source modifications. SLIC demonstrates that extending an existing operating system can be done in a manner that is protected from applications, enforced upon uncooperative applications, and efficient, while combining the development and testing advantages of user-level extensions with the performance of kernel extensions.

We have also examined the problems found in transparently extending operating system functionality, such as the asymmetric trust of the system call interface and implicit event information. Drawing from experiences with these problems, we presented a number of lessons that can be used by operating systems designers to provide interfaces which are conducive to interposition. Foremost is the imperative to maintain clear procedural barriers between operating system policy and mechanism. Additionally, to reduce the effort necessary in implementing an interposition mechanism, interfaces should be explicit and expose all information related to their events.

We believe that the techniques we have described in this paper can provide a substantial benefit to users of existing operating systems, enabling a viable third-party industry for developing and deploying operating system extensions. The resulting competition will stimulate innovation and increase the rate of technology transfer from operating systems research into production systems.

Availability

Current status and source code are available at `http://now.cs.berkeley.edu/Slic/`.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the 1986 USENIX Summer Conference*, pages 93–112, June 1986.

[2] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the Operating System at the User-Level: the Ufo Global File System. In USENIX, editor, *Proceedings of the 1997 USENIX Conference*, pages 77–90, January 1997.

[3] C. Amza, Alan L. Cox, Sandhya Dwarkadas, Peter Keleher, H. Lu, R. Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[4] Arindam Banerji, John M. Tracey, and David L. Cohn. Protected Shared Libraries — a new approach to modularity and sharing. In *Proceedings of the 1997* USENIX *Technical Conference*, pages 59–76, January 1997.

[5] B. N. Bershad, S. Savage, E. G. Sirer P. Pardyak, M. 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*, December 1995.

[6] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight Remote Procedure Calls. In *ACM Transactions on Computer Systems*, pages 37–54, February 1990.

[7] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. In *Proceedings of the Second International World Wide Web Conference*, pages 763–771, October 1994.

[8] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 27–37, October 1997.

[9] David D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, December 1985.

[10] Chaos Computer Club. CCC: Microsoft security alert. http://berlin.ccc.de/radioactivex.html, March 1997.

[11] Apple Computers. *Inside Macintosh, Macintosh Toolbox Essentials*. Addison-Wesley, 1992.

[12] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software—Practice and Experience*, 21(8):757–85, August 1991.

[13] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the 1996 ACM SIGMETRICS Conference*, 1996.

[14] D. R. Engler, M. F. Kaashoek, and Jr J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[15] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back, and Steven Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[16] Simon Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly and Associates, Sebastopol, CA, first edition, December 1994.

[17] Ian Goldberg, David Wagner, Randy Thomas, and Eric Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the Sixth USENIX Security Symposium*, July 1996.

[18] Wilson Hsieh, Marc Fiuczynski, Charles Garrett, Stefan Savage, David Becker, and Brian N. Bershad. Language support for extensible operating systems. In *Proceedings of the Workshop on Compiler Support for System Software*, February 1996.

[19] IBM Virtual Machine Facility /370 Planning Guide. Technical Report GC20-1801-0, IBM Corporation, 1974.

[20] Michael B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, December 1993.

[21] Yousef Khalidi and Michael Nelson. Extensible File Systems in Spring. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 1–14, December 1993.

[22] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, MA, 1991.

[23] Steven R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the 1986 Summer* USENIX *Technical Conference*, pages 238–247, 1986.

[24] Eduardo Krell and Balachander Krishnamurthy. COLA: Customized overlaying. In *Proceedings of the 1992 USENIX Winter Conference*, January 1992.

[25] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[26] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 229–243, October 1996.

[27] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 184–197, October 1997.

[28] John Ousterhout. Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proceedings of the 1990 Summer* USENIX *Conference*, pages 247–256, June 1990.

[29] Przemysław Pardyak and Brian N. Bershad. Dynamic binding in an extensible system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 201–212, October 1996.

[30] Dennis M Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, 1974.

[31] R. Sandberg, D. Goldberg, Steven Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985* USENIX *Technical Conference*, pages 119–130. Sun Microsystems, June 1985.

[32] Andrew Schulman, Raymond J. Michels, Jim Kyle, Time Paterson, David Maxey, and Ralf Brown. *Undocumented DOS*. Addison-Wesley, 1990.

[33] Margo I. Seltzer, Yasuhiro Endo, Christoper Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[34] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Issues in extensible operating systems. Technical Report TR-18-97, Harvard University, 1997.

[35] Emin Gün Sirer, Stefan Savage, Przemysław Pardyak, Greg DeFouw, Mary Ann Alapat, and Brian N. Bershad. Writing an operating system using Modula-3. In *Proceedings of the Workshop on Compiler Support for System Software*, February 1996.

[36] Christopher Small and Margo Seltzer. VINO: An Integrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard, October 1994.

[37] David C. Steere, James J. Kistler, and M. Satyanarayanan. Efficient User-Level File Cache Management on the Sun Vnode Interface. In *Proceedings of the 1990 USENIX Summer Conference*, pages 325–331, June 1990.

[38] Computer Emergency Response Team. Ghostscript Vulnerability. CERT Advisory CA-95.10, CERT, August 1995.

[39] Computer Emergency Response Team. Vulnerability in expreserve. CERT Advisory CA-96.19, CERT, August 1996.

[40] Computer Emergency Response Team. Vulnerability in Solaris admintool. CERT Advisory CA-96.16, CERT, August 1996.

[41] Computer Emergency Response Team. Vulnerability in WorkMan. CERT Advisory CA-96.23, CERT, October 1996.

[42] Computer Emergency Response Team. Vulnerability in talkd. CERT Advisory CA-97.04, CERT, January 1997.

[43] Marvin Theimer, K. Landtz, and David Cheriton. Preemptable Remote Execution Facilities for the V System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[44] Thorsten von Eicken, David E. Culler, Seth C. Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, May 1992.

[45] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[46] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 1–11, 1994.

[47] Nick Wingfield. ActiveX used as hacking tool. `http://www.news.com/News/Item/0%2C4%-2C7761%2C00.html`, February 1997.

[48] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–344, June 1974.

[49] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.

[50] G. Pascal Zachary. *Showstopper! The Breakneck Race to Create Windows NT and the Next Generation at Microsoft*. Macmillan, Inc., 1994.

[51] Sognian Zhou, Jingwen Wang, Xiaohn Zheng, and Pierre Delisle. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computing Systems. Technical Report CSRI-257, University of Toronto, 1992.