SLIC: An Extensibility System for Commodity Operating Systems

Douglas P. Ghormley Steven H. Rodrigues David Petrou Thomas E. Anderson

{ghorm,steverod,dpetrou,tea}@cs.berkeley.edu
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720

February 11, 1998

Abstract

Modern commodity operating systems are large and complex systems developed over many years by large teams of programmers, containing many hundreds of thousands of lines of code. Consequently, it is extremely difficult to add significant new functionality to these systems [15, 1, 40]. In response to this problem, a number of recent research projects have explored novel operating system architectures to support untrusted extensions; these include SPIN [7, 5], VINO [43, 39], Exokernel [19], Lipto [16], and Fluke [20]. Unfortunately, the architectures employed by these projects required substantial implementation effort and are not generally available in commodity systems. In contrast, by leveraging the technique of interposition, our extension mechanism requires only trivial operating system changes to enable 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, enforced upon non-cooperating applications, composable with extensions from other third-party sources, and significantly easier to develop than traditional kernel modifications.

We have designed and implemented a prototype extension mechanism called SLIC which utilizes interposition to efficiently insert trusted extension code into a commodity operating system kernel. We have used SLIC to implement a number of useful operating system extensions: 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 show that interposition on existing kernel interfaces can be accomplished efficiently.

1 Introduction

Modern commodity operating systems are extremely difficult and costly to modify. They are large, complex systems developed over many years by large teams of programmers and contain many hundreds of thousands of lines of code. In order to run on symmetric multiprocessors (SMPs), much of the code must be multi-threaded, compounding its complexity and requiring extensive revalidation for even small changes. It is not unusual for major releases of commodity operating systems to be riddled with flaws introduced while developing the new features of the release, 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 [15, 1, 40].

Although modifying commodity operating systems is complex and difficult, the need to do so remains. For example, there is a large catalog of value-add functionality which has not been widely deployed, in part because of the difficulty of modifying existing systems: load sharing [53], process migration [45, 15], fast communication primitives [6, 46], upcalls [13], distributed shared memory [30], user-level pagers [51], and novel schedulers [48, 17, 32]. 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 [12].

The goal of this work is to significantly simplify the process of evolving and extending existing commodity operating systems by enabling a large class of extensions—in particular, extensions which can manage global resources or enforce security guarantees. The ideal system which achieves this goal would include a number of characteristics. First, it would require few or no modifications to existing commodity operating systems in order to work efficiently. Second, multiple extensions from independent, third-party sources could be active simultaneously. Third, extensions would be protected from malicious applications and enforced upon non-cooperating applications. Finally, kernel extension developers would be able to make use of state-of-the-art development and debugging tools.

A consequence of accomplishing this goal would be to 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 transfered directly to end users without having to convince or wait for operating system suppliers to adopt the modifications. Furthermore, many CERT and CIAC security advisories which normally require a patch from the operating system vendor to fix would now be able to directly include a small extension to correct the flaws, reducing the window of vulnerability dramatically in some cases.

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

The first approach of re-engineering the operating system has been taken by a number of systems over the years which have attempted to reduce the cost of adding new operating system functionality by restructuring the operating system to be extensible. Systems built using this approach include Hydra [50], Mach [1], SPIN [7, 5], VINO [43, 39], Exokernel [19], Lipto [16], and Fluke [20]. While many of these systems have successfully demonstrated greatly reduced costs of adding new functionality, the cost of wholesale replacement of existing commodity operating systems prohibitive; for example, Microsoft has spent over $300M developing Windows NT [52]. Consequently, for the foreseeable future, it is unlikely that existing commodity operating systems will be re-engineered for extensibility in this manner.

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 [28, 37] is a prime example of this approach. However, applying this technique to make existing commodity operating systems more extensible would require substantially modifying and exposing all interfaces where additional functionality is desired, effectively re-engineering the majority of the operating system.

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

Prior attempts to use interposition as an extension mechanism suffer from significant limitations. Interposition Agents [25] leverages the Mach [1] system call redirection facility to transparently insert user-level extensions at the system call interface. However, extensions require application cooperation and are run unprotected in the application's address space. Interposition Agents therefore cannot enforce security guarantees or manage shared resources for competing applications. Software Fault Isolation (SFI) [47] can be used to rewrite application binaries to protect extensions running at user level from the application; unfortunately, it is difficult to apply SFI to arbitrary application binaries without prohibitive implementation complexity. Protected Shared Libraries [4] has the same capability as SFI with less software effort, but requires specialized hardware support that is not available on most architectures.

To investigate these issues, we have developed SLIC, a prototype system for efficiently inserting trusted extension code into existing operating system kernels 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 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 and a Linux 2.0 port is in progress.

We have used the SLIC prototype to implement a number of extensions that would have been significantly more difficult to accomplish by other means. One extension patches a security flaw publicized by CERT [10]. A second provides a configurable restricted process execution environment, while a third demonstrates the necessary hooks for an encrypting file system.

The rest of this paper is organized as follows. Section 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.

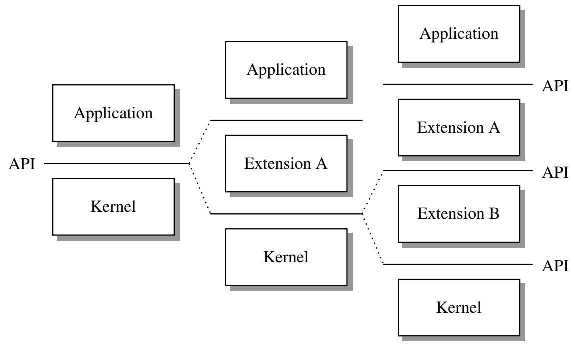

Figure 1: Interposing extensions on an interface. Solid lines indicate interfaces between components. Dotted lines illustrate the interposition of an extension on an interface. Function invocations on the original interface are intercepted and routed through extension code. In this diagram, the original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

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

Interposition enables *incremental* extensions in that extensions need only capture the events that they are interested in. Extensions do not have to handle all events crossing the interface boundary and they can leverage the functionality of the existing underlying interface; for example, an extension logging `fork()` system calls can use the underlying `write()` system calls of the operating system to store the log. This means that extension writers only have to implement the desired extension functionality, instead of 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 a second extension being added 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 any extensions.

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

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

However, interposition does have a number of 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. This can limit extension 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 [26].

Despite these limitations, the power and flexibility of interposition has led to its wide-spread use throughout modern computing systems. Forms of interposition can be found in the 'pipe' construct used in Unix shells, in the extensibility mechanisms of programming language systems [27], in distributed file systems such as NFS [38], in distributed shared memory systems such as TreadMarks [2], in World Wide Web proxy caches [8], and in MS-DOS terminate-and-stay-resident utilities and Macintosh toolbox extensions.

3 SLIC 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 of interposition to provide extensions with the following features:

Security: Extensions should be protected from malicious or faulty applications and should be enforced on non-cooperating applications. This enables extensions which manage shared resources, such as file systems, and extensions which enforce security guarantees. Note that SLIC assumes that extensions are trusted. Other research efforts have addressed the issues involved with supporting untrusted extensions [47, 5, 39, 31].

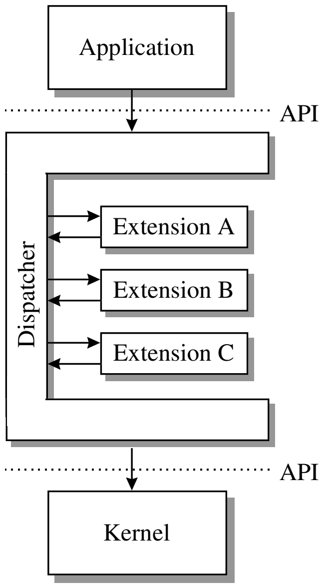

Figure 2: The basic design of SLIC. The dotted lines represent the interposed interface. Events crossing this interface are captured by a *dispatcher* and forwarded to one or more *extensions*.

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

Efficiency: Once development is complete, interposing extensions on the operating system should impose minimal overhead on the system. Per-extension overhead should be a few times the cost of a procedure call.

3.1 SLIC Architecture

SLIC is comprised of multiple *dispatchers* and *extensions* as well as various *support routines*. Dispatchers are responsible for intercepting system interface events and for routing those events to interested extensions. Extensions receive events from the dispatcher and implement new functionality. Support routines provide extensions with a simple, consistent interface to useful functionality such as memory allocation and synchronization primitives. These routines enable extensions to be portable across implementations of SLIC for various operating systems. Each dispatcher may provide additional support routines as appropriate for the interface; for example, a signal dispatcher may provide routines to determine the set of pending signals for a process.

3.1.1 Dispatchers

Each dispatcher is responsible for capturing events on a single system interface (such as system calls, signals, virtual memory events, or hardware interrupts). SLIC dispatchers use two different techniques to intercept interface events. For those interfaces which are implemented using jump tables, such as the system call, `vnode`, and virtual memory interfaces, SLIC records the original function address from the jump table and stores the address of its own interception routine. For procedural interfaces which are called directly from various locations in the kernel, such as the Solaris signal delivery and page replacement policy interfaces, dispatchers intercept events using binary patching. 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 from the interposed procedure are executed and control is returned to the original routine at the instruction following the binary patch. Using these techniques, SLIC dispatchers can intercept interface invocations for the cost of a procedure call.

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 inserted into the list in the order they were loaded.

Table 1 presents 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 pid 4191 and only calling `Slic_RegisterHandler()` to register a handler for the `open()` system call. Our signal dispatcher provides similar functionality, enabling filtering on process id and signal type.

Upon receiving an event, an extension has a number of options available: the extension can pass along the event unmodified by simply calling `Slic_IssueSyscall()`, the extension can modify the event parameters in `struct Slic_SyscallInfo` and then pass it along, or 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) by setting fields in the `Slic_ReturnInfo` structure. When an event is forced to return, extensions further down the call chain (including the kernel) never see the event for processing; the return value flows back up the chain in reverse order, allowing each extension to inspect or modify the returned value.

```
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 *syscallInfo,
                           Slic_ReturnInfo *returnInfo));
Slic_TraceProc(int pid, bool traceAllChildren);
Slic_UntraceProc(int pid, bool untraceAllChildren);

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

Table 1: A simplified version of the interface exported by the system call dispatcher to extensions.

Additionally, while processing one event, extensions may arbitrarily initiate other events 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.

On initialization, extensions register with one or more dispatchers using `Slic_RegisterExtension()`. The two predicates for the system call interface, a per-process tracing flag and a per-system call handler, are manipulated using the `Slic_TraceProc()`/`Slic_UntraceProc()` and `Slic_RegisterHandler()` functions. An extension is only invoked for a given system call event if the process which generated the system call is marked for tracing by this extension and the extension has a handler registered for this system call. Similar functionality is provided by the signal dispatcher.

3.1.2 Extensions

The SLIC architecture enables extensions to be structured in two ways, supporting a tradeoff between performance and ease of extension development. Extensions can be loaded as *user-level servers* or as *in-kernel extensions*, as shown in figure 3, or as a combination of these types.

Executing extension code as a *user-level server* enables 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 tools such as Purify [35] and Quantify [36]). The extension is protected against modification by user programs by virtue of running in a separate address space and is enforced on system applications by the dispatcher. The disadvantage of this approach is that invoking the extension from the dispatcher requires costly context switches and kernel-user boundary crossings. This organization is similar to that employed by micro-kernels such as Mach [1] and `/proc` systems.

An *in-kernel extension* is loaded directly into the kernel and is invoked directly from the dispatcher by a procedure call. When events are frequent, this organization has considerably better performance than the user-level approach. Extensions are protected against modification by user applications because they are in the protected kernel region of the address space. There are a number of limitations to this approach: the kernel is not protected from malicious or faulty extension code and there is no support for user-level development tools. A number of methods of protecting the kernel from extensions have been explored by others [34, 24, 41, 39, 31]. By supplying extensions with the same interface, whether at the user level or in the kernel, extensions can be safely developed at the user level and then inserted into the kernel. This enables development in a safe environment without sacrificing the potential for good performance.

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.

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 UltraSPARC workstations; work is in progress for a Solaris page replacement policy dispatcher and a port to the Intel 80*x*86 version of Linux 2.0, an operating system whose internals share no heritage with Solaris. All SLIC components — dispatchers, extensions, and support routines — are dynamically loaded by the system administrator 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()` and `sigaddqa()` functions; the signal dispatcher intercepts signals by binary patching these functions at run-time. To enable the run-time modification of these functions, one line of Solaris source code was

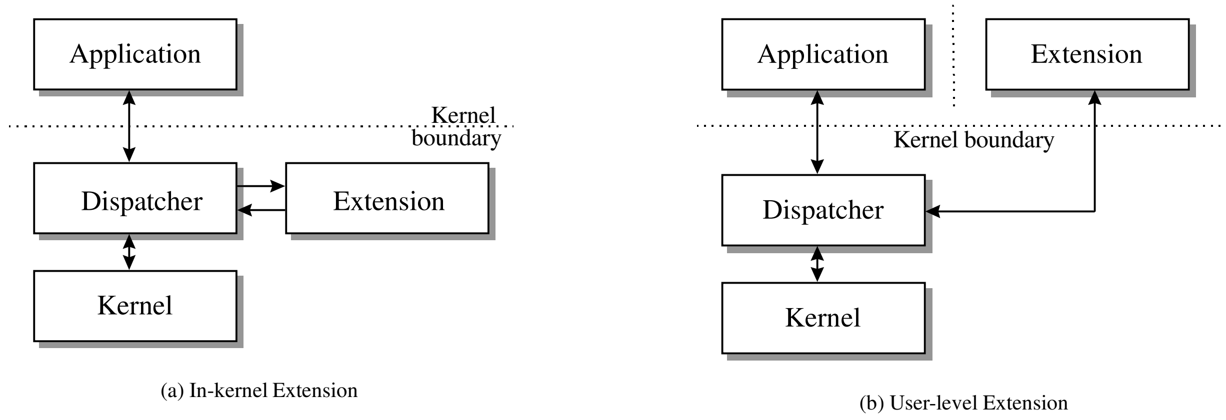

Figure 3: Organization alternatives in SLIC. In (a), extension code is located within the kernel's address space, while in (b), extension code is run as a separate user level process. Dotted lines represent protection domain boundaries. Passing the event to a user level extension (as in (b)) involves multiple context switches and protection boundary crossings.

changed to make the kernel text image writable by SLIC dispatchers.

The current implementation catches events within the kernel, rather than at the machine level (interrupts); for example, system calls are caught at the `sysent` table rather than upon execution of the `trap` instruction. While the two approaches are conceptually similar, catching events within the kernel considerably simplified the implementation of our prototype.

3.2.1 System Call Dispatcher

System calls transfer data in two forms: 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 in-memory application buffers, however, is more tricky since the data is stored in the application's address space.

This leaves a window of vulnerability between the time when the extension inspects or modifies the buffer and the time when the underlying operating system executes the system call. Other threads in the application could alter exploit this window of vulnerability to alter the buffer, effectively passing information directly to the kernel and avoiding the extension. This violates the requirement that extensions be enforced on uncooperative or malicious applications. To prevent this situation, memory buffers that are inspected or modified by an extension must be protected from applications. Placing such modified buffers in the kernel provides the necessary protection but introduces another problem. During system call execution, the Solaris kernel performs protection checks on accesses to memory buffers, requiring that those buffers be located in the application's address space. These checks are performed in the kernel's `copyin()` and `copyout()` routines and are normally necessary to prevent malicious applications from reading or overwriting sensitive kernel data.

Our solution to this problem is to maintain a per-thread list of valid, in-kernel extension buffers and to interpose on the `copyin()` and `copyout()` routines, permitting accesses to these regions when appropriate. When an extension allocates an in-kernel buffer to a thread, SLIC adds the allocated region to that thread's valid list. When the `copyin()` or `copyout()` routines are invoked, the interposed code checks the accessed address against the thread's valid list. If the address is a valid extension buffer, then 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 routines which will perform the normal protection checks. The list of valid extension buffers is maintained on a per-thread basis to ensure that applications cannot illegally read or modify buffers allocated to other applications.

3.3 Micro-benchmarks

We have measured the overhead imposed by SLIC on system calls and signals. All micro-benchmark numbers reported are from a 167MHz UltraSPARC running Solaris 2.5.1.

There are three micro-benchmarks; we ran each operation one million times and averaged the results. The first micro-benchmark performs a `getpid()` call, which in Solaris is essentially a null system call. This micro-benchmark measures the raw overhead of the system call dispatcher, but does not invoke the modified copy routines. In order to quantify the overhead resulting from our modification to the copy routines, the second micro-benchmark performs a `sigprocmask()` system call. This system call involves a memory copy of 16 bytes from a kernel data structure into user space. The `kill()` micro-benchmark 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. This micro-benchmark measures the overhead of the signal mechanism and the cost of interposing on the signal mechanism.

We tested our micro-benchmarks with various configurations of SLIC: unloaded (i.e., an unmodified system), loaded and active with no extensions, and loaded and active with multiple extensions. The extensions used for these measurements are null extensions which merely catch events and reissue them. For system call events, after an extension reissues a system call, the extension also receives the return value from this system call. The results are presented in Table 2. The implementation of user-level extensions for the signal interface is not yet complete.

The basic overhead of the SLIC dispatchers is between 300 and 600 nanoseconds for system calls and roughly 9 microseconds for signals. The incremental overhead of loading additional system call extensions is between 2.5 and 5.2 microseconds, while signal extensions cost between 4 and 9 microseconds. The difference between the `getpid()` and `sigprocmask()` columns is that getpid() does not use any memory arguments, while `sigprocmask()` invokes the modified copy routines. One of the primary sources of per-extension overhead is the allocation of register windows in the UltraSPARC. The current implementation allocates a register window for every extension that wishes to examine the return value from a system call. Measurements indicate that system call extensions which do not capture return values, and thus do not require a register window, add an average of only 220 nanoseconds to the dispatcher overhead.[1]

4 Extending the Operating System

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 one of the system tools while awaiting a patch, and the encryption file system and restricted execution environment would have required substantial kernel source code modification to achieve similar functionality and performance. Further, without SLIC, changes to add these features to the existing kernel would likely be *ad hoc* rather than a general solution that can be leveraged for future extensions.

[1] A note to reviewers—we are still in the process of evaluating the performance of the dispatchers.

4.1 CERT Advisory extension

The Computer Emergency Response Team (CERT) of Carnegie Mellon University's Software Engineering Institute regularly provides the Internet community with information regarding system security problems. Whenever possible, these advisories include information on how to resolve the problem reported. However, due to the lack of extensibility in existing systems, frequently this advice is to completely disable the insecure feature [10, 9, 11]. Though operating system vendors do respond to these advisories by supplying patches, those patches can take weeks to become available [12]. Since CERT does not have access to source code for many systems, they are unable to directly provide patches for these problems. However, using SLIC, many of these advisories could be accompanied by small extensions which would resolve the problem.

To demonstrate this, we have implemented an extension to patch a recent security hole discovered in the Solaris `admintool` [10] which allowed unprivileged users to delete arbitrary files in the system in certain circumstances. Our 25 line extension monitors file operations, denying the operations which cause the problem. Continued use of the `admintool` is maintained.

4.2 Encryption File System

In a distributed file system, maintaining file security 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 [21]. However, stand-alone encryption tools and libraries can be time consuming or cumbersome to use and are not easily integrated with existing application binaries. 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, a much easier approach is to implement a single encryption extension which can be interposed 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 [39]. The extension watches for `open()` and `creat()` system calls of files with a particular suffix and then records the process id 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. Issues of key management and encryption algorithm are orthogonal to demonstrating the proof-of-concept that interposition provides a simple yet powerful means of transforming file sys-

	getpid()	sigprocmask()	kill()
Unmodified system	2.82	3.90	80.83
SLIC, dispatcher loaded, no extensions	3.08	4.53	89.09
SLIC, dispatcher loaded, $1 \times$ null extension (disabled)	3.22	4.44	88.99
SLIC, dispatcher loaded, $1 \times$ null extension	8.27	9.69	99.28
SLIC, dispatcher loaded, $2 \times$ null extension	10.79	12.28	103.27
SLIC, dispatcher loaded, $3 \times$ null extension	13.37	14.84	108.01
SLIC, dispatcher loaded, null extension at user level	57.45	62.53	

Table 2: Micro-benchmark performance of SLIC. All numbers are in microseconds and are averaged over one million iterations. The third row reports the performance of a single null extension when not intercepting events. The `getpid()` and `sigprocmask()` tests were run with the system call dispatcher loaded while the `kill()` test was run with the signal dispatcher. The `getpid()` test is effectively a measure of a null system call; the `sigprocmask()` test measures the overhead of SLIC's interposition on the kernel copy routines. The `kill()` benchmark measures the time required for a process to signal itself. The implementation for user-level extensions of the signal interface is not yet complete.

tem 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 trust the program being run. For example, programs downloaded from untrusted sources such as the Internet may actually be Trojan horses designed to steal or destroy information [49, 14]. In addition, there are cases in which the user trusts the program, but not the data being processed, as in the case of helper applications used by web browsers to display various data formats. Input data could potentially exploit bugs in helper applications to insert viruses into the system, a process similar to that used by the Internet Worm `fingerd` attack [18].

The tracing facility of the standard `/proc` file system is one method that is commonly used to construct a restricted execution environment. Potentially insecure system calls are captured and then selectively denied or altered [22]. The `/proc` approach suffers from a number of shortcomings. First, the Solaris `/proc` file system (as of version 2.5.1) only allows system calls to be denied with `EINTR` as the error code, a code normally used to indicate that the attempted operation was interrupted and should be retried. Returning this code causes many applications to loop endlessly rather than receiving a permission violation error. Second, the `/proc` file system cannot trace programs which are marked `setuid`; in Solaris, this includes programs such as `crontab`, `ps`, and `ping`. Finally, 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. 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 [22] 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()` can be disabled. Any attempts by the application to perform a proscribed operation results in the extension returning an `EPERM` error. The extension monitors only the subset of system calls necessary to maintain the security guarantees, providing low 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.

Special attention is given to connection to the X Windows System. Malicious applications running under the X Windows System can request X events from other windows on the user's machine, potentially snooping sensitive data such as passwords. Our restricted execution environment extension solves this problem by forcing the untrusted application to connect to an independent X server such as `Xnest`[2]. Our extension can check socket connections and dynamically reroute them to the `Xnest` server instead of the standard X server. Because it runs under SLIC, this extension cannot be circumvented and thus maintains its security guarantees. The restricted environment used for benchmarking denies 45 system calls outright (for example `chown()`), and performs security checks, such as checking the path or file access permissions (for example `rmdir()`), for 21 additional system

[2] `Xnest` is part of the standard X11R6 distribution.

calls.

4.4 Performance

To evaluate the impact of these extensions on system performance, we ran the extensions under three benchmarks: the Modified Andrew Benchmark [23, 33], a TEX compilation of a 262-page (760KB) document, and a `gcc` compilation of `emacs-19.34` without support for X Windows. The Modified Andrew Benchmark consists of multiple phases which (i) create directory subtrees, (ii) copy files, (iii) search file attributes via `find` commands, (iv) search files for a text string via `grep`, and (v) compile files. While the benchmark fits entirely in the file cache of modern systems and is therefore not useful for measuring file system performance, this fact will expose the overhead imposed by SLIC. The TEX benchmark was chosen to be representative of a document processing workload. Lastly, we chose the `gcc` benchmark because it performs significant I/O. Table 3 reports some relevant statistics for each benchmark.

Table 4 presents the results of running the benchmarks on each extension as well as on all extensions simultaneously. To decrease variability due to disk latency, all data files for the benchmarks were placed in a memory-mounted `/tmp` file system. The TEX benchmark shows very little change in performance due to SLIC, as expected from the relative infrequency of system calls listed in Table 3. We believe that variance in the measurements accounts for the fact that the TEX benchmark appears to run faster under the Restricted Execution Environment than with no extensions loaded. Though SLIC imposes a certain amount of overhead on applications, the last line in Table 4 illustrates that the much of the overhead experienced by the benchmarks is due to the SLIC infrastructure, a cost which is only paid once; the per-extension overhead is small for common workloads.

5 Evaluating Interposition

This section describes our experiences in 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 interposing on the virtual memory mechanism and page replacement policy for Solaris, the scheduler interface for FreeBSD 2.2.5R, and our experiences in an initial port of SLIC to Linux 2.0.

The problems that we have encountered can be divided into four categories:

1. The asymmetric trust mechanisms of the system call interface;
2. The lack of explicit information in the interfaces we interposed on;
3. An incomplete decomposition of functionality in the system; and
4. Miscellaneous implementation issues.

5.1 Asymmetric Trust Mechanisms

The system call interface is an untrusted interface; the kernel views all data provided by user applications as potentially malicious and performs security checks to ensure that user processes cannot access secure kernel state. System call extensions must take the same distrustful view of application-supplied data. For example, when an application issues an `open()` system call, it provides the kernel with the address of a buffer containing the file name. Before actually reading data from this buffer, the kernel validates that the buffer is located in the process' address space. However, extensions such as the CERT patch need to perform their own security checks. Security would be violated if an application were able to modify the file name after the CERT extension validated the file name but before the kernel actually performed the `open()`. To prevent this, the CERT extension must first copy the file name into a buffer that the user application cannot access (for instance, a kernel buffer); once validated by the CERT extension, the address of this secure buffer is then passed to the kernel to perform the `open()` operation.

However, because the CERT extension is interposed on the system call interface, the kernel views the extension as part of the untrusted user application. Consequently, the kernel's protection mechanisms will notice that the buffer supplied by the CERT extension is inaccessible to the process and the system call will be aborted with an error.

Resolving this problem requires either modifying or disabling the kernel's security checks for these buffers. SLIC interposes on the kernel's `copyin()` and `copyout()` routines to do this. When the routines are invoked, SLIC checks the target address against a per-thread list of valid secure buffers. If the address in question is a valid extension buffer, then the kernel's `bcopy()` routine is invoked to copy the data to its final destination. If the address does not point to a valid extension buffer for this thread, then control is passed back to the original `copyin()` or `copyout()` routine which will succeed or fail as normal.

5.2 Lack of Explicit Information

Many of the interfaces in today's operating systems are not fully *explicit*: some information is not passed directly as an

	Procs	Total System Calls	System calls Caught CERT		Encrypt		Rexec	
AFS	471	40500	5760	14%	2732	7%	9246	23%
TEX	3	2457	126	5%	850	35%	188	8%
gcc	379	140656	43720	31%	11031	8%	47888	34%

Table 3: 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 of those system calls which are caught by each extension.

	AFS Time (s)	AFS Overhead	TEX Time (s)	TEX Overhead	gcc Time (s)	gcc Overhead
Baseline	15.81		6.36		158.30	
No extensions	16.65	5%	6.69	5%	164.14	4%
CERT	17.51	11%	6.65	5%	163.83	3%
Encrypt	17.45	10%	6.42	1%	166.87	5%
Rexec	17.14	8%	6.48	2%	166.35	5%
CERT + Encrypt + Rexec	17.92	13%	6.78	7%	168.14	6%

Table 4: Benchmark performance on sample extensions. All measurements were run on a 167MHz UltraSPARC running Solaris 2.5.1. Each column for each benchmark contains both the total elapsed run time in seconds and the percent slowdown. "Baseline" represents a machine without any SLIC dispatchers or extensions loaded. "No extensions" represents the system call dispatcher loaded, but no extensions. The rows labels "CERT", "Encrypt" and "Rexec" (Restricted Execution Environment) present the benchmark elapsed times with a single extension loaded. The last line presents benchmark performance with all three extensions interposing simultaneously.

argument to the event but rather is stored in global data structures. For example, when a system call is performed, the process id of the calling process is not part of the system call invocation. System call error conditions are also not explicit; they are instead stored directly in the kernel's process structure and loaded into the user's registers upon return to user mode. The Solaris virtual memory system relies heavily on global data structures to determine information such as page ownership.

Our solution to this scattering of information has been to develop utility functions which provide extensions with access to important information such as process identifiers. A number of these utility functions are non-trivial, and have to walk through internal operating system data structures such as the process table. A well-formed interface would make this information readily accessible when an event is raised.

While the system call interface is generally quite conducive to interpose on (despite its lack of explicit information), the `ioctl()` call is a notable exception. Originally designed as a way to manage arbitrary devices, `ioctl()` has evolved into the generic method of extending Unix functionality. An `ioctl()` call can pass in arbitrary memory buffers, which may contain pointers to other memory buffers; the structure of each buffer is defined by the particular device driver. This means that an extension cannot know, in advance, how to handle the arguments to 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()`. We are currently experimenting with more elaborate interposition on the `copyin()` and `copyout()` routines to capture all device reads and writes from user space, as a means of identifying `ioctl()` memory buffers.

5.3 Separation of Policy and Mechanism

Extensible operating systems papers often argue that the separation of policy and mechanism is essential for maintaining extensibility. Our experiences agree with this contention. While experimenting with extending the scheduler interface in FreeBSD 2.2.5R, we found that the FreeBSD scheduler routine, `cpu_switch()`, implements both the policy of selecting the next process to run as well as the actual context switch mechanism. To enable interposition on the scheduler policy, we relocated the code which selects the next process to run into a separate routine, creating a procedural interface which can be interposed on.

A variation of this separation problem occurs with the Linux routines for accessing data in user space. These routines (`getuser()` and `putuser()`) are normally inlined, making interposition by binary patching extremely difficult. By simply modifying the Linux source to disable inlining of those copy routines, we again create a procedural interface that can be interposed on.

5.4 Miscellaneous Issues

Our method of using binary patching to intercept procedural invocations was simple to implement. However, intercepting procedural interface events using binary patching requires a writable kernel text. Unfortunately, Solaris is loaded such that the kernel text is read-only. By modifying a single line of Solaris source, we were able to make the text writable. In earlier versions of Solaris, a system configuration file enabled kernel writability, but this functionality has recently been removed.

SLIC dispatchers and extensions often need a way of recording state that persists across event invocations. For instance, the system call dispatcher needs to keep track of which processes are marked for tracing and a virtual memory extension may need to store per page information. The traditional place to store this information is in the process table or the page structures, but in many operating systems, the size and organization of these structures are compiled into system utilities and other kernel modules such as device drivers. Consequently, we currently implement shadow structures of the kernel's process and thread structures to store state for SLIC dispatchers and extensions.

Extensions operate with the privileges of the calling thread when they are invoked. Consequently, an extension cannot write to files owned by root or signal root-owned processes, except when invoked from a root process.

5.5 Guidelines for Kernel Developers

Given these experiences, there are a number of guidelines which can be formulated for making operating system interfaces conducive to interposition. First, when an interface delineates a boundary between trusted and untrusted code, the protection checks of untrusted data should be isolated in separate functions (which Solaris does). This separation allows the security checks to be cleanly interposed on and augmented without affecting other interface functions.

Second, it is important to distinguish between mechanism and policy, and to maintain this distinction even in the compiled binary. This enables extensions to replace system policies while still leveraging the mechanisms of the system. Although this solution may impact performance slightly, the costs involved with most mechanisms and policy decisions in today's production operating systems considerably outweigh the relatively low costs of a procedure call.

Third, a full set of functions should be provided to manage the non-explicit data of an interface. For instance, Solaris provides a function to set the error value for a system call, but does not provide a function to clear that error, which some extensions may need to do. Currently dispatchers provide such functionality in utility functions, but doing so requires an understanding of many details of the system. Providing these functions in the underlying kernel will make dispatcher implementations simpler and more robust.

Finally, a simple method should be provided to enable dispatchers to write to the kernel code image. This is necessary to enable interposition on procedural interfaces using binary patching.

6 Related Work

There has been a considerable amount of recent work [16, 43, 5, 19, 20] that has focused on building highly-extensible operating systems. Because these systems have built new operating systems kernels and new kernel structures, they have not focused on the problem of extensibility for existing systems. Of these systems, SPIN [5] and VINO [43, 39] 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, rather than enabling SLIC-style extensibility on existing interfaces. SPIN and VINO also aggressively focus on ensuring kernel protection from untrusted extensions, SPIN by using a type-safe language [42, 24], and VINO through software fault isolation [47] and in-kernel transactions [39]. SLIC assumes trusted extensions.

Interposition Agents [25] demonstrated that it is useful to construct interposition extensions in terms of the high-level abstractions of the interposed interface, rather than in terms of the low-level events crossing that interface. The interposition technology used in [25] was the standard Mach system call redirection facility which bounces system calls to extensions linked into an application's address space. This mechanism has two disadvantages relative to SLIC. First, extensions are not protected from applications and thus cannot implement security extensions or share data between distrustful applications. Second, the multiple protection boundary crossings limit the performance of the system. Our interposition technology enables high-performance interposition that is both enforced upon, and protected from applications, enabling a significantly larger class of extensions. In principle, the toolkit presented in [25] could ported to our interposition platform and could further simplify the process of extension development.

COLA [29] 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 interposition technology used in [25].

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 is primarily used for adding new interfaces to a system, although they could be combined with the interposition mechanisms used in SLIC to enable modification of existing interfaces. The PSL protection technique relies on hardware features of the IBM RS/6000 architecture, while the principles in SLIC are generally applicable across a variety of operating system platforms.

7 Future Work

Work on SLIC is proceeding in a variety of directions. We are focusing on adding additional dispatchers, such as for the page replacement policy and file system interfaces of Solaris 2.5.1. In addition, we are working toward making the base system portable to other platforms, and on developing a large number of useful extensions.

An area of active research is in identifying and managing conflicts between extensions. While interposition enables stacking of extensions, the mechanism does not guarantee that the extensions will be compatible with each. Prior experiences with with conflicts among MS-DOS terminate-and-stay-resident (TSR) utilities and Macintosh toolkit extensions indicate that a method of identifying and managing conflicts among incompatible extensions is sorely needed.

A Linux 2.0 port of the base SLIC system is in progress. Because the base system encapsulates a significant amount of functionality, extensions written for the Solaris version of SLIC should port with few or no modifications to the Linux version.

We are also exploring the design space of useful extensions to be built with SLIC. Among these are transparent remote execution, a single system image in a network of workstations, extending the semantics of NFS to include full cache-coherency for simultaneous access [44] and supporting scheduler activations [3] through interposition on signals or directly on the scheduler.

8 Conclusion

This paper has examined the utility of interposition as a mechanism for adding extensibility to production operating systems. 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 extensions through interposition in Solaris with minimal kernel source modifications. SLIC demonstrates that extending an existing operating system can be done efficiently and securely, enabling a larger class of extensions than previous work in this area.

We have also examined the problems found in transparently extending operating system functionality, such as the asymmetric trust found across the system call interface. 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, in order to reduce the effort necessary in implementing an interposition mechanism, extension interfaces should be explicit and expose all information related with an event.

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

Availability

SLIC is implemented on Solaris 2.5.1 and a port to Linux 2.0 is in progress. Current status and source code is 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] 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.

[3] Thomas Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *ACM Transactions on Computer Systems*, pages 53–79, February 1992.

[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, Anaheim, CA, 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] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemysław Pardyak, Stefan Savage, and Emin G. Sirer. SPIN—An Extensible Microkernel

for Application-Specific Operating System Services. Technical report, University of Washingtion, 1994.

[8] 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.

[9] Vulnerability in expreserve. CERT Advisory CA-96.19, CERT, August 1996.

[10] Vulnerability in Solaris admintool. CERT Advisory CA-96.16, CERT, August 1996.

[11] Vulnerability in WorkMan. CERT Advisory CA-96.23, CERT, October 1996.

[12] Vulnerability in talkd. CERT Advisory CA-97.04, CERT, January 1997.

[13] David D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, December 1–4 1985.

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

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

[16] Peter Druschel, Larry L. Peterson, and Norman Hutchinson. Beyond Micro-Kernel Design: Decoupling Modularity and Protection in Lipto. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 512–520, June 1992.

[17] 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.

[18] Mark W. Eichin and Jon A. Rochlis. With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988. In *Proceedings of the 1989 IEEE Symposium on Research in Security and Privacy*, 1989.

[19] 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.

[20] 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.

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

[22] 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.

[23] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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, Atlanta, GA, 1986. Sun Microsystems.

[29] Eduardo Krell and Balachander Krishnamurthy. COLA: Customized overlaying. In *Proceedings of the Winter 1992* USENIX *Technical Conference*, San Francisco, CA, January 1992. AT&T Bell Laboratories, USENIX Association.

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

[31] 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.

[32] 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*, October 1997.

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

[34] Przemysław Pardyak and Brian N. Bershad. Dynamic binding in an extensible system. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 201–212, Seattle, WA, October 1996. University of Washington, USENIX Association.

[35] Pure Software, Inc. *Purify User's Guide*, 1996.

[36] Pure Software, Inc. *Quantify User's Guide*, 1996.

[37] David S. H. Rosenthal. Evolving the vnode interface. In *Proceedings of the 1990 Summer* USENIX *Technical Conference*, pages 107–117, Anaheim, CA, June 1990. Sun Microsystems.

[38] 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, Dallas, TX, June 1985. Sun Microsystems.

[39] 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.

[40] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Issues in extensible operating systems. Publication pending from IEEE Press, 1997. Available from `http://www.eecs.harvard.edu/~vino/vino/papers/acm-96.ps`.

[41] Emin Gün Sirer, Marc Fiuczynski, Przemysław Pardyak, and Brian N. Bershad. Safe dynamic linking in an extensible operating system. In *Proceedings of the Workshop on Compiler Support for System Software*. University of Washington, February 1996.

[42] 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*. University of Washington, February 1996.

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

[44] V. Srinivasan and Jeffrey C. Mogul. Spritely NFS: Experiments with cache-consistency protocols. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 45–57, Litchfield Park, AZ, December 1989. ACM.

[45] 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.

[46] Thorsten von Eicken, David E. Culler, Steh 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.

[47] 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.

[48] 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, November 1994.

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

[50] 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.

[51] 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.

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

[53] 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.