

User-Level Sandboxing: a Safe and Efficient Mechanism for Extensibility

Richard West and Jason Gloudon

Computer Science Department
Boston University
Boston, MA 02215
{richwest,jgloudon}@cs.bu.edu

Abstract

Extensible systems allow services to be configured and deployed for the specific needs of individual applications. This paper describes a safe and efficient method for user-level extensibility that requires only minimal changes to the kernel. A sandboxing technique is described that supports multiple logical protection domains within the same address space at user-level. This approach allows applications to register sandboxed code with the system, that may be executed in the context of any process. Our approach differs from other implementations that require special hardware support, such as segmentation or tagged translation look-aside buffers (TLBs), to either implement multiple protection domains in a single address space, or to support fast switching between address spaces. Likewise, we do not require the entire system to be written in a type-safe language, to provide fine-grained protection domains. Instead, our user-level sandboxing technique requires only paged-based virtual memory support, and the requirement that extension code is written either in a type-safe language, or by a trusted source.

Using a fast method of upcalls, we show how our sandboxing technique for implementing logical protection domains provides significant performance improvements over traditional methods of invoking user-level services. Experimental results show our approach to be an efficient method for extensibility, with inter-protection domain communication costs close to those of hardware-based solutions leveraging segmentation.

1. Introduction

Fundamental to the design of any general purpose operating system is the need to provide trusted services to applications. These services are responsible for the predictable and safe management of hardware resources and

abstractions. However, the services typically implemented by general purpose systems are often ill-suited to the specific needs of many applications. For example, a real-time application is not well served by a scheduling policy that does not consider the timely and predictable execution of processes. Similarly, a web server [24] may benefit from its own buffer cache algorithm that over-rides the default caching and paging policy [3]. This has motivated researchers to study various system designs that support extensibility [32, 16, 2, 31, 6, 8, 14], thereby allowing services to be tailored to the requirements of the applications they support [4, 15].

While extensible systems have desirable properties, they have posed at least three conflicting challenges: (1) how to guarantee *efficient* execution of the extension code that modifies or adds functionality to the system, (2) how to ensure the *safety* of extensions that could otherwise violate the integrity of the system, or applications themselves, and (3) how to provide support for extensibility without significant modification to the standard interfaces offered by the system. One approach that is arguably most efficient is to allow extensions to be linked into the address space of the operating system kernel, thereby allowing them to be invoked by direct function calls within the kernel protection domain, or via system calls from user-space. The downside is that application-specific code is placed in a trusted address space that may jeopardize system correctness and/or violate security guarantees.

Various approaches have been proposed to guarantee the safety of kernel service extensions [36]. These approaches rely on techniques such as type-safe languages [2, 27, 28, 9, 22], *sandboxing* [30, 31], proof-carrying codes [29], or hardware-support [6, 26]. Notwithstanding, many researchers have argued that kernel extensions are inappropriate for a number of reasons, beyond the sophisticated techniques necessary to enforce safety. One argument against extensible kernels is that an extension interface must be provided, in addition to a standard API, that affects com-

patibility, interoperability, and evolution of OS implementations [10]. Similarly, supporters of micro-kernels [13, 26, 20] argue that only a core set of base abstractions should be part of a kernel, while more complex abstractions and (extensible) services should be implemented in user-space. This approach provides a clear separation of concerns and, hence, ease of addition and modification of services without affecting the kernel.

Unfortunately, there are costs associated with the implementation of extensible services at user-level. Overheads are typically incurred as a result of communication via the trusted kernel, as well as scheduling and switching between address spaces that isolate services. In fact these overheads, associated with the communication between user-level services in separate address spaces, have caused major problems for the efficient design of micro-kernels. This has led to the consensus that micro-kernels are inherently non-portable if they are to be implemented efficiently, implying that special hardware support is necessary [26].

Regardless of whether or not service extensions are implemented in the kernel or at user-level, they need to be isolated in logical protection domains that cannot adversely affect the behavior of the system, or applications. That is, the range of memory addresses accessible to application-specific service extensions must be restricted. While features such as segmented memory can be used to provide logical protection domains within a single address space, such hardware support is not common on all processors (except a few such as the Intel x86 [21]). However, most hardware platforms for general purpose systems do support page-based virtual memory, that requires each protection domain to exist in its own address space, mapped to a range of physical addresses using private page tables. It is on these platforms, that there are costs associated with switching and communicating between logical protection domains, since the processing context has to be changed each time we switch to a new address space.

In summary, it is desirable to allow the services of a system to be customized for the specific needs of applications. Supporting user-level extensibility has the benefit that kernel interfaces remain largely unchanged, and there is a clear division between extension and kernel code. However, providing a safe and efficient mechanism for user-level extensibility is still a challenge, especially on commercial off-the-shelf (COTS) systems that primarily support protection at the granularity of page-based address spaces.

1.1. Motivation and Contributions

This paper is motivated by the desire to implement extensible services at user-level in a manner that is safe, efficient, and requires minimal changes to the underlying kernel. We show how it is possible to achieve these goals using a *user-*

level sandboxing technique, that enables COTS systems to be extended for the specific needs of applications. In fact, our approach places no specific requirements on the underlying OS structure. As a consequence, it is possible for our technique to implement micro-kernel services, interposition agents [23, 17], virtual machines [18, 33] and entire OSes in a sandboxed region above a kernel that is, say, monolithic.

Our approach differs from other solutions in that it neither relies exclusively on hardware (e.g., Palladium [6]) nor software (e.g., software fault isolation [30], or Java) support. Instead, we combine both hardware and software features to implement logical protection domains within page-based address spaces that may be accessed efficiently via a trusted kernel. We show how this technique can be used to safely implement application and system service extensions, in a manner similar to the way new services and extensions are implemented in micro-kernels, extensible, and library-based systems [12]. Our technique allows application and system developers to map service extensions into a sandboxed memory region shared by all process address spaces running in the system. The sandboxed region leverages page-based hardware protection while extension code itself is either written by a trusted source, or in a type-safe language.

By supporting service extensions at user-level, there are several advantages. Most notably:

- it is possible for such code to leverage libraries that would be unavailable within the kernel,
- there is no need for custom interfaces as extensions can instead leverage existing system call interfaces,
- it is possible to rapidly prototype code that would otherwise cause system failure, and
- extensions can be developed in a manner similar to regular application code without awareness of kernel internals.

We show, with minimal modifications to an existing operating system (specifically, a monolithic x86-based Linux system), the efficient implementation of logical protection domains at user-level that have the performance benefits of approaches requiring special hardware support. Using the proposed techniques, we show how to efficiently extend the behavior of an existing system at user-level without the traditional costs of communicating between logical protection domains.

Using a fast method of upcalls [7], we show how our sandboxing technique for implementing logical protection domains provides significant performance improvements over traditional methods of invoking user-level services. We believe our approach is a promising method for the safe and efficient implementation of user-space application and system service extensions. It is appropriate for COTS systems, requiring minimal changes to the kernel.

Experimental results show that using our approach to *in-*

terpose [23, 17] code between an application and the underlying system results in lower impact on application performance than traditional methods. Likewise, our technique for implementing (and communicating between) logical protection domains in a single address space has similar performance to hardware-based solutions, such as the “small spaces” work on the Pentium [34]. Finally, we compare various techniques to access sandboxed service extensions. On a Pentium 4 processor, we can safely switch from the kernel to a sandboxed extension function in 11000 cycles, compared to 46000 cycles if we invoke a user-level extension in a private address space that is not currently active.

The following section describes our user-level sandbox technique in more detail. This is followed by Section 3 that evaluates the performance of our approach on a Linux x86 platform. Related work is then discussed in Section 4. Finally, conclusions and future work are described in Section 5.

2. User-Level Sandboxing

Overview: The basic idea of *user-level sandboxing* is to modify the address space of all processes, or logical protection domains, to contain one or more shared pages of virtual addresses. The virtual address range shared by all processes provides a sandboxed memory region into which extensions may be mapped. Under normal operation, these shared pages will be accessible only by the kernel. However, when the kernel wishes to pass control to an extension, it changes the privilege level of the shared page (or pages) containing the extension code and data, so that it can be executed with user-level capabilities. This prevents the extension code from violating the integrity of the kernel. However, the extension code itself can run in the context of *any* user-space process, even one that did not register the extension with the system. There is potential for malicious extension code to modify the memory area of a running process or extract sensitive data values. To guard against this, we require extension code registered with the system to either: (1) be written in a type-safe language that enforces memory protection, or (2) be written by a trusted programmer. In the latter case, encryption could be used as a means to authenticate extensions.

While others have argued that type-safe languages [9, 22, 27, 28] incur more costs than hardware-based protection schemes [6], our approach only relies on language-level type-safety for extension code mapped to the sandbox region. All other application and system-level code can be written in non-type-safe languages. This differs from the approach of SPIN [2] and JavaOS that require all software objects to be type-safe. Additionally, type-safe languages offer the potential for finer granularities of memory protec-

tion than most hardware can accommodate.

2.1. Hardware Support for Memory-Safe Extensions

Our approach assumes that hardware support is limited to page-based virtual memory. A series of caches, most notably one or more untagged translation look-aside buffers (TLBs) is desirable but not necessary. This minimum hardware requirement is met by most general purpose processors made today. More specialist hardware methods of implementing logical protection domains to accommodate extension code include the use of processors with tagged TLBs, or combined segmentation and paging units. Tagged TLBs provide a fast way to switch between protection domains mapped to separate address spaces, by storing the virtual-to-physical address translations of these address spaces in non-overlapping regions of a dedicated hardware cache. Alternatively, hardware lacking tagged TLBs but supporting segmented memory has been used to isolate these logical protection domains in different memory segments.

Tagged TLBs have the advantage that they do not need to be flushed and reloaded when switching between address spaces (e.g., during a process context switch), unlike untagged TLBs that only cache address translations for an unspecified virtual memory region. In contrast, processors such as the Intel x86 have untagged TLBs but employ both segmentation and paging units. Specifically, the x86 processor uses segmentation hardware to convert between *logical* and *linear* addresses, and paging hardware to translate between linear and *physical* addresses; untagged instruction and data TLBs are used only to cache linear-to-physical translations. The advantage with an architecture such as the x86 is that protection domains can be mapped to separate memory segments restricted to specific ranges of linear addresses. Switching between protection domains mapped to different ranges of linear addresses simply involves switching the active segment, by changing base and limit values on addresses, without affecting page table entries cached in a TLB.

While segmentation hardware can isolate logical protection domains from one another, it is not supported by many processors other than the x86 (and, in a similar form, by the HP PA-RISC). Likewise, tagged TLBs are not common on the most popular processors for general purpose systems. This has meant that many general purpose systems implement course-grained protection domains using only paging hardware that is fairly universal amongst hardware vendors, even in the embedded system world¹. To emphasize the point, x86-based systems such as Linux limit the use of

¹e.g., the StrongArm SA1110 processor found in some PDAs employs page-based virtual memory support.

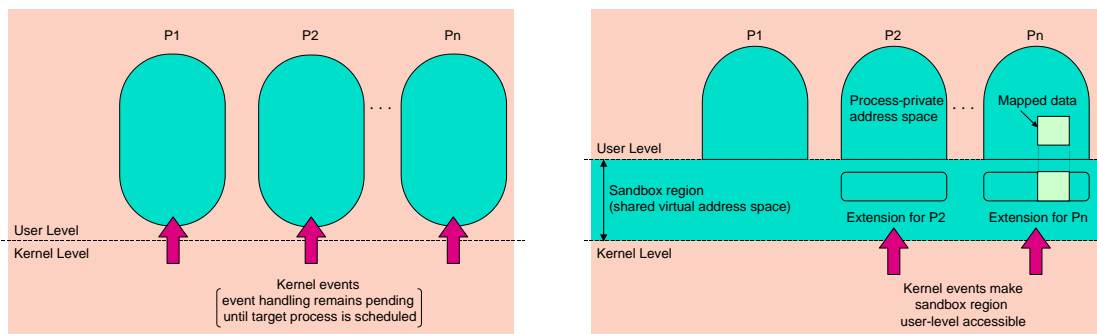


Figure 1. (a) Traditional view of logical protection domains, each mapped to a separate address space. (b) Each process address space has a shared virtual memory region, or sandbox, into which extensions are mapped. Kernel events, delivered via upcalls to sandbox code, are handled in the context of the current process, thereby eliminating scheduling costs. Part of the sandbox itself is switched between kernel and user privilege levels.

segmentation to isolate kernel and user-level protection domains. Process and thread address spaces are defined in terms of linear addresses within user- or kernel-level segments. This means that switching between one address space and another requires a switch between page tables used for linear address translation.

On many processors, the costs of switching between protection domains mapped to different pages of virtual (or linear) addresses, requires switching page tables stored in main memory, and then reloading TLBs with the necessary address translations. Such course-grained protection provided at the hardware-level is becoming more undesirable as the disparity between processor and memory speeds increases [34]. This is certainly the case for processors that are now clocking in the gigahertz range, while main memory is accessed in the 10^8 Hz range. In practice, it is clearly desirable to keep address translations for separate protection domains in cache memory as often as possible. User-level sandboxing avoids the need for expensive page table switches and TLB reloads by ensuring the sandbox is common to all address spaces.

2.2. Implementation Details

We have implemented user-level sandboxing on a Linux x86-based system, with a few small changes to the kernel. These changes are required to: (1) create a shared sandbox region, (2) support secure mapping of sandboxed extensions, (3) allow access to restricted sandboxed memory regions from conventional process address spaces, and (4)

invoke extension functions from within the kernel. For the most part, our approach is not restricted to Linux. However, where necessary, we describe the system-specific features required for user-level sandboxing to work. The user-level sandboxing implementation requires a few additional interface functions over those provided by the traditional system call interface. However, these interface functions are invoked via ioctls, avoiding the need for new system calls. As a result, kernel changes are implemented in a manner similar to device drivers, and can be included in kernel-loadable modules, with minimal modification to the core kernel.

Logical Protection Domains for Extension Code: Traditional operating systems provide logical protection domains for processes mapped into separate address spaces, as shown in Figure 1(a). With user-level sandboxing (Figure 1(b)), each process address space is divided into two parts: a process-private memory region and a shared virtual memory region. The shared region acts as a sandbox for mapped extensions. Technically speaking, the sandbox is further divided into *public* and *protected* areas, as explained later. Kernel events, delivered by upcalls to sandbox code, are handled in the context of the current process, thereby eliminating scheduling costs.

Sometimes it is important for a process to exchange data with extensions registered in the sandbox. As a result, we allow controlled access to a region of sandbox addresses by both code in a process-private region and code in the sandbox itself. An ioctl function, registered with the kernel, called `allocate_mapped_data()`, maps a region of the sandbox into a process-private address space, as in Fig-

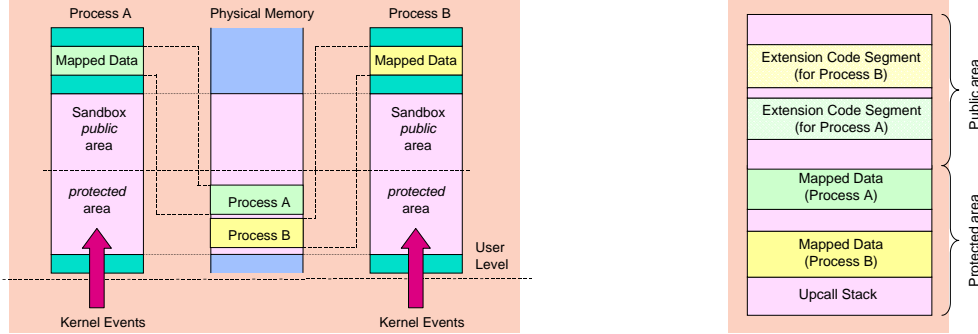


Figure 2. (a) Sandboxes common to all processes are mapped to the same physical address ranges. Applications can map pages from their private address space into the sandbox for reading and writing data. (b) An example sandbox layout for two processes.

ure 1(b). This mapped region is read- and write-accessible by the calling process and sandbox extensions that can reference the corresponding range of addresses. This data-sharing scheme relies on hardware page protection, and so `allocate_mapped_data()` allocates memory on page boundaries. In our implementation, pages are $4KB$ in size. Now, an application process, for which type-safety is not guaranteed, can arbitrarily rewrite pointers in this memory shared with the sandbox, circumventing the controls on memory access by extension code. References to these mapped regions are consequently passed to extension code as arrays of character or integer values, making type and pointer safety in extensions immune to arbitrary writes by application code.

Sandbox Regions: The sandbox consists of two $4MB$ regions of virtual memory that are identically mapped in every address space to the same physical memory (as shown in Figure 2(a)). The two regions are backed by physically contiguous RAM and are adjacent to each other for implementation convenience. These regions employ the page size extensions supported by the Pentium processor and each occupy one 4 megabyte page directory entry². The normal memory management interfaces are not allowed to allocate memory from these regions as they must be shared by multiple users.

One sandbox region is permanently assigned read and execute permission at both user- and kernel-level and acts as a *public* area. This region is marked as a global page using the global flag supported by Pentium Pro and more recent IA-32 [21] processors. This prevents the page directory entry for this page from being invalidated when a context switch occurs. The other region is permanently assigned read-write permission at kernel-level but by default

the region is inaccessible at user-level. We refer to this region as the *protected* area. The region can be made accessible to user-level by toggling the user/supervisor flags of its page directory entry and invalidating the relevant TLB entry via the `INVLPG` instruction.

Sandbox/Upcall Threads: A design decision we had to make was whether or not to allow threads to execute in the sandbox. If we allowed code in the sandbox to invoke system calls, it could be possible for an extension registered by one process to block the progress of another process. For example, if process p_i registers an extension e_i that is invoked at the time process p_j is active, it may be possible for e_i to affect the progress of p_j by issuing ‘slow’ system calls. One solution is to prevent extensions from issuing system calls. However, one advantage of user-level extensibility we wanted to support was the ability to leverage libraries, and many such library functions make system calls. It would be overly restrictive to prevent user-level extensions from issuing system calls. Instead, all system calls issued by an extension could be made non-blocking. Again, this would be restrictive and may require extension code to be linked against special libraries. Moreover, not all system calls can be made non-blocking.

Finally, we chose to support threads of execution in the sandbox. At first this seems counter-intuitive, as it requires threads to have their own stacks and they must be scheduled. In a system such as Linux, threads are treated almost like conventional processes when it comes to scheduling. That is, both have identifiers and task structures for their state information, but threads merely share their address space with the parent process. Since sandbox threads execute in *any* process context, essentially they are inexpensive to schedule.

A sandbox-bound thread of execution is created by a user-level process using the `create_upcall()`

²The 32-bit x86 processor uses a two-level paging scheme, comprising page directories and tables.

interface. `create_upcall()`, like the POSIX `pthread_create()` produces a new thread of control sharing the credentials and file descriptor tables of the caller. The thread produced by `create_upcall()`, however, does not possess a conventional hardware-based address space. Instead, sandbox threads execute using the page tables of the last active address space.

Mapping Code into the Sandbox: The existence of a shared sandbox requires the modification to the page tables and address spaces of all created processes (when they are first ‘forked’). Applications can register handlers and extensions that are mapped into this sandbox (see Figure 2), where they may be executed in the context of any process, since all processes will have page tables that can resolve virtual addresses of instructions and data in this memory area.

The code segment of a sandbox-bound thread is loaded into the read-only public area of the sandbox, while the writable data segments are loaded into the protected area. A modified version of the `insmod` routine, from the GNU `modutils` suite is used to map code into the sandbox. As both areas are mapped to the same addresses in the page tables of every process, the sandbox-bound thread always has access to its code segment. However, before the sandbox-bound thread may access its data segments at user-space, the user/supervisor flag for the protected area must be appropriately set at the cost of the necessary TLB invalidate operation and the subsequent TLB reload. If necessary, this is done when context switching to the sandbox thread. When the process whose page tables were used by the sandbox thread is again scheduled, the user/supervisor flag must be reset before the process regains control of the CPU at user-level. This is necessary to keep malicious processes from gaining access to the protected sandbox area.

Building for the Sandbox: As sandbox-bound threads do not have conventional address spaces, they are unable to use certain system interfaces related to memory management without modification. Some of the affected interfaces include `brk()`, `mmap()` and `shmget()`. These interfaces are used to fulfill a variety of needs: `brk()` serves to allocate and deallocate process-private virtual memory (for the purposes of dynamic memory allocation), while `shmget()` allocates shared memory segments. Likewise, `mmap()` can allocate either process-private or shared virtual memory as well as providing memory-mapped file I/O. We are in the process of modifying these functions as part of our own version of a trusted `libc` library, to ensure memory allocation occurs in the correct address ranges.

Fast Upcalls: Traditionally, signals and other such kernel event notification schemes [1, 25] have been used to invoke actions in user-level address spaces when there are specific kernel state changes. Unfortunately, there are costs associated with the traversal of the kernel-user boundary, process context-switching and scheduling. The aim is to im-

plement an upcall mechanism with the speed of a software trap (i.e., the mirror image of a system call), to efficiently vector events to user-level where they are handled by service extensions, in an environment that is isolated from the core kernel.

Operating systems such as Linux that leverage hardware protection to separate user- and kernel-address spaces do not support conventional trap gates to user-level. General protection faults occur when attempting to trap to a ‘ring of protection’ that is less critical than the kernel. However, architectures such as the Intel IA-32 support instructions such as `SYSENTER` and `SYSEXIT` that can be used in conjunction with Model Specific Registers (MSRs) [21] to allow fast transitions between kernel and user-level address spaces. On the IA-32 architecture, where there are four rings of protection, we use these instructions to transition between rings 0 (the kernel privilege level) and 3 (the user privilege level). While this is not a particularly portable approach, it is possible to replace these instructions with ‘activation records’ placed on a kernel stack, that trick the hardware into thinking it is returning to user-level. This approach is adopted by Palladium [6] and is the typical way most kernels return control back to user-level after servicing a system call. `SYSENTER` and `SYSEXIT` were first featured in the Pentium II processor. We chose to use these instructions as they avoid unnecessary memory references and protection checks, since they require transition between ring 0 and ring 3.

To avoid the problem of raising kernel events for upcall handlers (or extensions) when no user-level process is running, all upcall handlers utilize a private stack in the sandbox (see Figure 2(b)). Note that it is possible for a kernel thread, having no user-level context outside the sandbox, to be executing at the time an upcall is activated. Additionally, we must schedule the target thread for the execution of upcall handlers before switching privilege levels. It should be noted that sandbox extensions cannot be invoked other than via the trusted kernel.

Potential Protection Problems: When an upcall event is issued from the kernel, the mechanism will modify an entry in the current process’s page table to allow user-level access to the sandbox *only while the upcall event is being processed*. Preemption and signal handling during the execution of code in the sandbox must be disabled. Allowing preemption may cause reentrancy problems (e.g., if another process runs and an upcall occurs again), while conventional signal handling can provide a ‘trap door’ into the sandbox for malicious users. For example, if a signal is delivered to the current process while executing sandbox code, the sandbox memory region is open to read-write access via the signal handler. Minor changes to the kernel simply delay delivery of signals until extensions have completed execution and the sandbox protected area is reset to

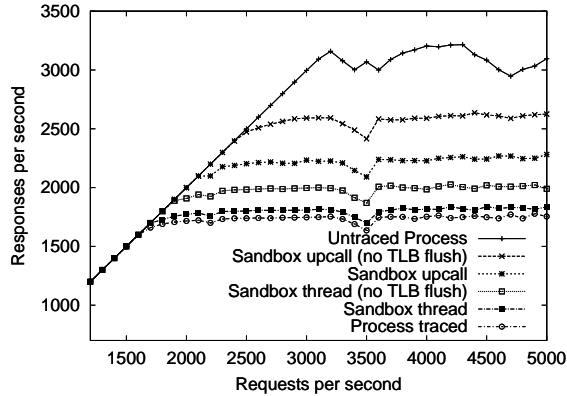


Figure 3. The performance of a tthttpd web server while tracing its system calls using a variety of mechanisms. This shows how interposition code mapped into a user-level sandbox reduces the impact of system call tracing, compared to traditional methods such as ptrace.

the supervisor privilege level.

3. Experimental Evaluation

A series of experiments were conducted to assess the effectiveness of a user-level sandboxing implementation in a modified Linux 2.4.9 kernel running on 1.4 GHz Pentium 4 based systems connected by a Gigabit Ethernet network. These experiments investigate the relative costs of implementing interposition agents, the performance of intra-address space extensions and the effects of TLB working set sizes on user-level sandboxing.

3.1. Interposition

Interposition agents [23, 17] introduce user code between the operating system interface and applications, in order to modify or replace the services that the operating system provides. The Linux 2.4 kernel series introduced extensions to ptrace(2) to facilitate system call interception at user-level. The ptrace mechanism incurs a large overhead in the form of several context switches per system call. This significantly reduces the performance of applications under interposition that make frequent system calls. To show how the sandbox can reduce interposition overheads, a number of minimal interposition agents were implemented which read the system call number of each system call made by an unmodified tthttpd 2.20c web server under a range of HTTP request loads. The HTTP requests were generated from another host over a Gigabit Ethernet network using httpperf. The same file was targeted in each request, which was made with a timeout of 1 second. The average rate of successful

responses was recorded over 30000 requests. Three types of interposition agents were compared:

- A kernel scheduled thread in a traditional process address space using ptrace ('Process traced')
- A kernel scheduled thread in the sandbox using ptrace ('Sandbox thread')
- An upcall handler based agent executing in the sandbox ('Sandbox upcall')

Figure 3 shows the relative performance of these agents, compared to the situation where the web server runs untraced ('Untraced Process'). In this experiment, the interposition code introduces overheads, but we want to investigate the interposition method that minimizes these overheads. Running an agent in the sandbox yields consistent improvement in the web server performance over the traditional address space based agent. A more significant improvement is shown by the upcall handler based agent, which is invoked from the kernel when a system call is made without the overhead of having to schedule a thread. Measurements were also taken for the sandbox based agents where the sandbox was left open to user space, so that no TLB flushes were performed when context switching between the web server and the agent (i.e., the 'no TLB flush' cases in Figure 3). These results show that with the existing ptrace interface, the sandbox can be used to reduce interposition overheads, and that with the appropriate interface additional gains can be made.

3.2. Inter-Protection Domain Communication

To investigate the effects of working set size on the effectiveness of sandbox-based extensions, a number of IPC ping-pong experiments similar to those conducted in the

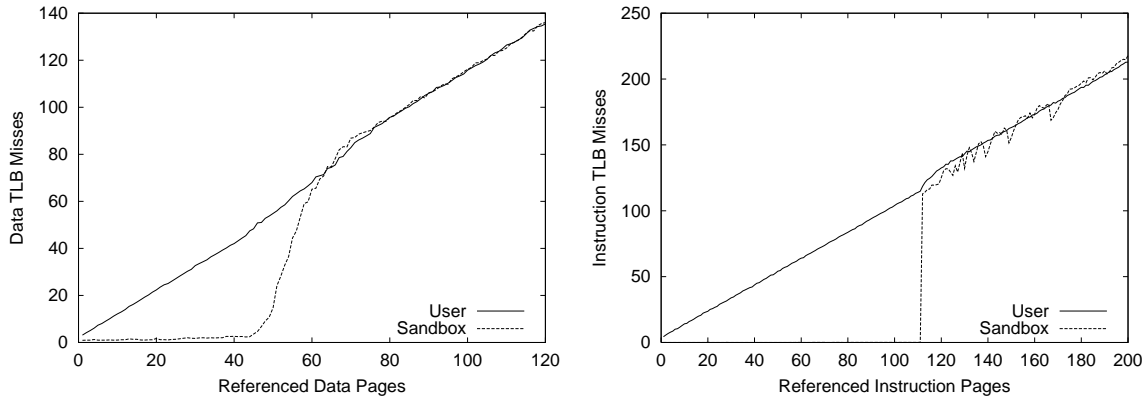


Figure 4. Effects of working set sizes in terms of (a) data, and (b) instruction pages on the number of TLB misses, for inter-protection domain communication. The ‘User’ case is for traditional inter-process communication, while the ‘Sandbox’ case shows communication costs between a process and a sandboxed protection domain.

“small spaces” work [34] were carried out. These experiments also consider the effects of both instruction and data TLBs, found on the x86 architecture. The Pentium 4 processor has a 64 entry data TLB and an 128 entry instruction TLB for address translation.

Two threads exchange four byte messages over connected pipes. One thread simulates an application thread in a traditional address space with a configurable instruction and data TLB working set. The second thread acts as an extension running either in a separate full address space or in the sandbox. The “application” thread fills some number of TLB entries, sends a message to the “extension” thread, and reads a reply message. The application thread then accesses its previously referenced pages. The extension thread, which has a small fixed TLB size, is executed either in the sandbox or in a traditional address space. To simulate various data TLB sizes, the application thread reads 4 bytes of data from a series of memory addresses spaced 4160 byte apart. To simulate instruction TLB sizes, the application thread performs a series of relative jumps to instructions spaced 4160 bytes apart. These spacings were chosen to avoid cache interference effects. The TLB miss counts were obtained using the Pentium 4 CPU performance counters.

Figure 4(a) shows the data TLB working set of the application thread is maintained for up to approximately 45 entries when the extension thread is mapped into the sandbox. At this point the combined data TLB demands of the operating system, the application and the extension no longer fit the 64 entries available on the Pentium 4 and each page access incurs a TLB miss. Note that for the extension thread in a traditional address space, every data page access after the IPC ping-pong incurs a TLB miss regardless of the work-

ing set size, as all TLB entries have been purged. That is, the untagged TLBs of the x86 are flushed and reloaded for every context switch between different address spaces.

The instruction TLB entries of the application thread are also preserved when the extension is located in the sandbox, as shown in Figure 4(b). No instruction TLB misses occur until the working set approaches 110 entries, which is close to the available 128 TLB entries. Thereafter, the number of instruction TLB misses are close for both extension types, with the sandbox extension finally causing slightly more TLB misses. These results are similar to those in the “small spaces” work that uses the segmentation features of the x86 to implement multiple logical protection domains *within* a single address space. This shows that our user-level sandbox technique can achieve inter-protection domain communication performance similar to approaches based on specialist hardware features such as segmentation.

Additionally, Figure 5(a) shows the communication latency remains lower with the sandbox extension even when the data TLB miss rates are similar. Likewise, in Figure 5(b), the pipe latency is considerably lower for the sandboxed extension, until the instruction TLB is filled. Once the working set is no longer able to fit in the available instruction TLB entries, the resulting latency for the sandbox extension is still slightly lower than the traditional address space extension.

3.3. Web Server Performance Using Dynamic Content Requests

Further experiments were carried out to evaluate the performance of applications composed of traditional address

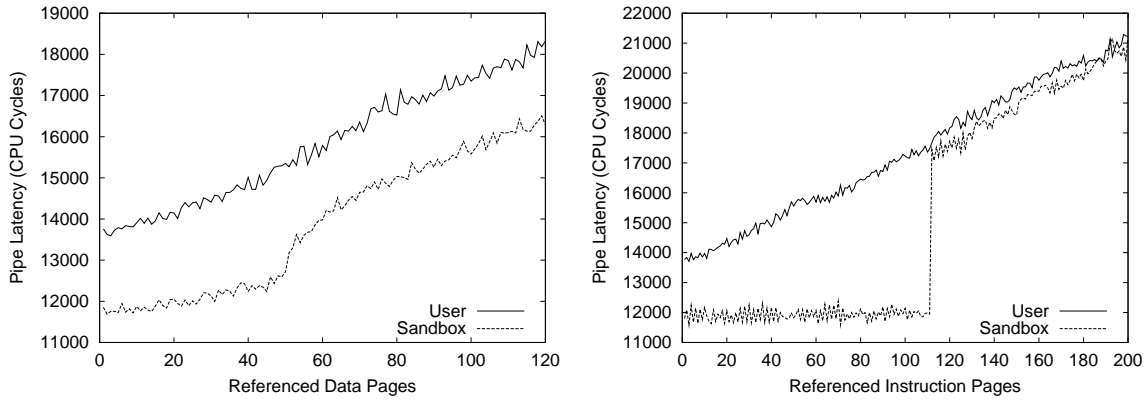


Figure 5. Latency of communication via a pipe between two protection domains, as a function of working set sizes in terms of (a) data, and (b) instruction pages.

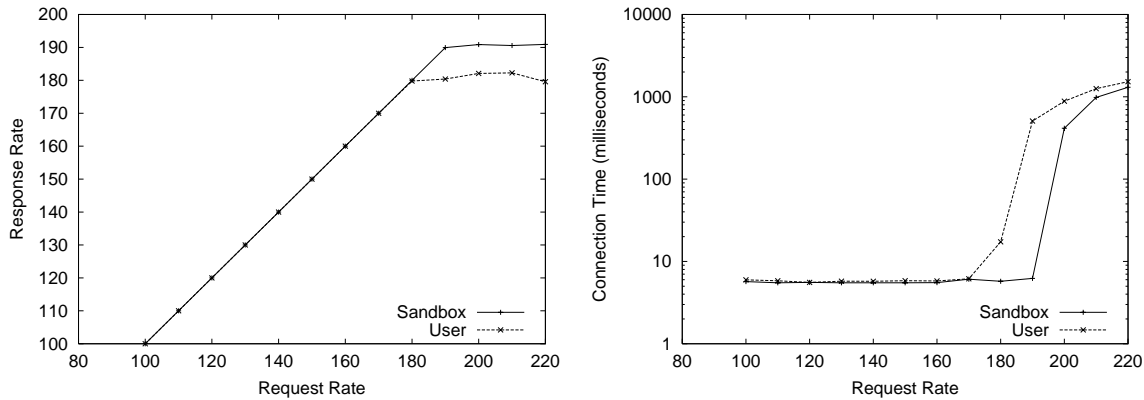


Figure 6. Performance of an unmodified Apache 2.0.44 web server handling dynamic content requests with the help of FastCGI processes mapped to both separate address spaces ('User' case) and sandboxed memory regions ('Sandbox' case).

space processes extended with sandbox-based code. An unmodified Apache 2.0.44 web server was configured to support FastCGI, an interface between web servers and external processes that generate dynamic content. Apache was configured to communicate over a local UNIX domain socket with a multi-threaded FastCGI process to satisfy HTTP requests coming from another host. On each request the FastCGI process reads a 36 Kilobyte XML file from the filesystem, transforming it into a 20 Kilobyte HTML response. Each request is generated by httpperf with a 5 second timeout. Figure 6 shows the performance of application, when FastCGI threads are running in a separate address space from the Apache server process (the 'User' case), and when FastCGI threads are executing within the sandbox (the 'Sandbox' case). As can be seen, the maximum response

rate is improved when FastCGI threads are mapped to the sandbox. Similarly, the average request connection time remains lower for a larger request rate when FastCGI threads execute in the sandbox.

In a similar experiment, dynamic content was provided by Java Servlets running within an Apache Tomcat 4.1 servlet container. Here, the web server was configured to use multiple threads running either in the sandbox or in a traditional address space, communicating with the servlet container process over TCP sockets. The performance of the web server-servlet engine was measured while running the web server in the sandbox as well as in a traditional address space. In contrast to the FastCGI experiments, the Java servlet experiment shows reduced performance for a sandbox-based Apache server, with slightly higher connec-

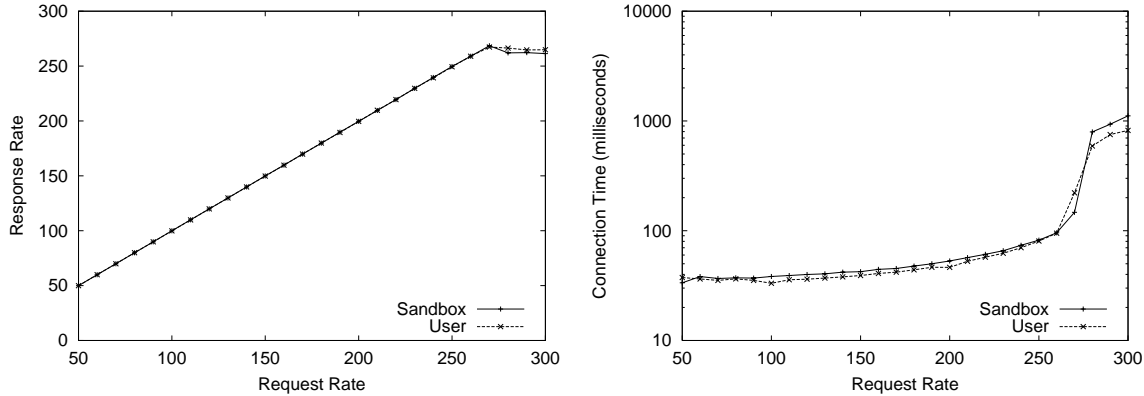


Figure 7. Performance of an Apache 2.0.44 web server handling dynamic content requests with the help of Java servlets. The ‘User’ case shows Apache mapped into a conventional user-level process address space, while the ‘Sandbox’ case shows Apache mapped into the sandbox.

Operation	Cost in CPU Cycles
Upcall including TLB flush/reload	11000
TLB flush and reload	8500
Raw upcall	2500
Signal delivery (current process)	6000
Signal delivery (different process)	46000

Table 1. Microbenchmarks taken on a 1.4GHz Pentium 4, 512 Megs RAM. Cycles given above are based on the time stamp counter.

tion times at almost all request loads. This is shown in Figure 7. We attribute the differences in these results to the larger instruction TLB working set size of the Apache HTTP server, compared to the FastCGI extensions.

In summary, these experiments show that when the working sets of logical protection domains do not exceed the TLB limits, fast inter-protection domain communication is possible with our user-level sandboxing method. This is possible without the need for special hardware support, such as segmentation.

3.4. Microbenchmarks

Table 1 presents a number of microbenchmarks that point to the effectiveness of using our fast upcalls method, for invoking sandbox code. We compare our approach to conventional signals. The complete upcall cost includes the CPU cycles required to go from kernel space to a user-space upcall handler function. This includes the costs of flushing the sandbox data area TLB entry, placing arguments on the upcall stack, performing a SYSEXIT and executing the user-level prologue of the upcall handler function.

The TLB flush and reload time dominates the overall upcall cost, while the SYSEXIT instruction cost, referred to as the raw upcall cost, accounts for less than a third of the elapsed cycles. The signal costs measure the overheads of delivering a signal to user space from the kernel within the same address space context as well as between different address spaces. The costs of delivering a signal within the same address space is much lower than the cost of an upcall, as no hardware protection overheads are involved, but once an address space switch and scheduling operation are involved the costs of delivering a signal from kernel to a user-space process are over 4 times the cost of a full upcall. Note that the measured cost of delivering a signal to a different process involves making that process the highest priority, so it is guaranteed to be scheduled next.

4. Related Work

There have been a number of related research efforts that focus on OS structure and extensibility, safety, and service invocation. Extensible operating systems research [32] aims

at providing applications with greater control over the management of their resources. In contrast, microkernels such as Mach [13] offer a few basic abstractions, while moving the implementation of more complex services and policies into application-level components. By separating kernel- and user-level services, microkernels introduce significant amounts of interprocess communication overhead. This has caused microkernels to fall out of favor despite substantial reductions [26] in communication costs.

Other OS approaches, such as the Exokernel [12], try to efficiently multiplex hardware resources among applications that utilize library operating systems. Applications are built by linking with untrusted code that implements traditional operating system abstractions at user-level using low-level primitives. Resource management is thus delegated to library operating systems, which can be readily modified to suit the needs of individual applications, resulting in good extensibility. One technique applied in Exokernels is downloading code in the form of ‘Application Specific Safe Handlers’ [35] that perform low latency, application-specific processing of network messages, thereby avoiding the overhead of application scheduling and dispatching. In common with the Exokernel approach, user-level sandboxing enables services and extensions to be linked into process address spaces, along with library code they may use. The difference is, our approach is aimed at extending existing systems, by slight modifications to the kernel, as opposed to employing an entirely new (albeit small) trusted kernel.

In contrast, SPIN [2] is an extensible operating system that supports extensions written in the Modula-3 programming language. This language provides type-safety and memory protection, by enforcing interface contracts between code modules. Extensions signed by the trusted system compiler are deemed safe and may be loaded into the kernel address space at runtime. Interaction between the core kernel and SPIN extensions is mediated by an event system, which dispatches events to handler functions in the kernel, without the overhead of kernel/application boundary crossing. By providing handlers for events, extensions can implement application-specific resource management policies with low overhead. Similar to SPIN, our approach requires extensions are written in a type-safe language if they are not written by a trusted source. The difference is that our extensions execute with user-level privileges, and may run on slightly modified COTS systems.

A transaction-based approach to system extensibility is employed by the VINO [11] operating system. VINO supports system extensions known as *grafts* that are written in C++. Since C++ is not type-safe and memory protection is an issue, grafts are run in the context of transactions, so that the system can be returned to a consistent state if a graft is aborted. As with all the above approaches, our work differs in that it is aimed at providing user-level extension support

without the need for a special-purpose OS.

Other extensible systems research includes the DEIMOS system [8], SLIC [16], Palladium [6] and approaches that leverage interposition [23, 17]. DEIMOS is novel in that it does not define a kernel in the strict sense. Instead, it uses a configuration manager to encapsulate, load and configure (on demand) traditional kernel functions and application-specific services in the form of modules. SLIC is interesting in that it leverages interposition to support extensibility of commodity operating systems. This has a lot in common with our user-level sandboxing approach. In SLIC, a series of dispatchers are responsible for intercepting system events on a particular interface, and for routing those events to appropriate extensions that may reside in kernel or user-space. However, user-level extensions are encapsulated in separate process address spaces that incur expensive scheduling and context-switching costs when invoked by dispatchers. In contrast, user-level sandboxing allows extensions to co-exist at user-level in a shared address space, avoiding expensive context-switch overheads. Finally, Palladium leverages both segmentation and multiple rings of protection to support both user- and kernel-level extensions. Interestingly, Palladium reorganizes application and extension code so that extensions are always located in a less privileged ring of protection than the code that invokes their services. However, this method is primarily targeted at x86-based systems and relies on hardware support for protection. User-level sandboxing does not require segmentation and multiple rings of protection.

Another area of research related to ours has focused on service invocation, kernel event notifications [1, 25] and upcalls [7, 19]. Much of this work is concerned with the way to trigger user-level services or handlers due to some condition or event in the kernel. The ‘Kqueue’ work [25] found in FreeBSD is similar in concept to the kernel event notification work of Banga et al [1]. Both pieces of work provide an efficient means of delivering events to user-space, that are typically triggered by I/O state changes and traditionally handled by heavyweight constructs such as `poll()` and `select()`. However, Kqueues provide a general framework that not only supports event notifications relating to I/O (via socket and file descriptors), but also events traditionally associated with signals. While both mechanisms alleviate the costs associated with traditional methods of event delivery, they do not offer upcalls mechanisms akin to the mirror image of a system call. Our approach, to use constructs such as `SYSENTER` and `SYSEXIT` on the Pentium processor, is similar to the way Palladium [6] allows user- and kernel-level extensions to be activated from a more privileged ‘ring of protection’. While `SYSENTER` and `SYSEXIT` provide a fast way to switch between ring 0 and 3 on the Pentium, there is no reason why our user-level sandboxing technique cannot leverage Palladium’s more general

‘activation record’ method. Palladium requires modification of the kernel stack of the target process address space, to spoof the hardware into believing it is *returning* to a less privileged protection domain to execute extension code. We plan on leveraging this approach for hardware platforms that support multiple rings of protection but lack the convenient SYSENTER and SYSEXIT instructions.

Finally, observe that our work is not to be confused with user-level resource-constrained sandboxing [5], by Chang, Itzkovitz and Karamcheti. Their work focuses on the use of sandboxes to enforce quantitative restrictions on resource usage. They propose a method for instrumenting an application, to intercept requests for resources such as CPU cycles, memory and bandwidth. As a result, they are able to control the allocation of such resources in a predictable manner. The emphasis of our work is to develop fine-grained protection domains at user-level for the execution of extensions, regardless of which address space is active at the time extension code is invoked.

5. Conclusions and Future Work

Extensible systems allow services to be configured and deployed for the specific needs of individual applications. While extensibility is desirable it poses at least three conflicting challenges: (1) how to guarantee *efficient* execution of the extension code that modifies or adds functionality to the system, (2) how to ensure the *safety* of extensions that could otherwise violate the integrity of the system, or applications themselves, and (3) how to provide support for extensibility without significant modification to the standard interfaces offered by the system.

This paper describes a safe and efficient method for user-level extensibility that requires only minimal changes to the kernel. We describe a *sandboxing* technique that supports multiple logical protection domains within the same address space at user-level. This approach allows applications to register sandboxed code with the system, that may be executed in the context of any process. Our approach differs from other implementations that require special hardware support, such as segmentation or tagged TLBs, to either implement multiple protection domains in a single address space, or to support fast switching between address spaces. Likewise, we do not require the entire system to be written in a type-safe language, to provide fine-grained protection domains. Instead, our user-level sandboxing technique requires only paged-based virtual memory support, and the requirement that extension code is written either in a type-safe language, or by a trusted source.

Experimental results show that using our approach to *interpose* [23, 17] code between an application and the underlying system results in lower impact on application performance than traditional methods. Likewise, our tech-

nique for implementing (and communicating between) logical protection domains in a single address space has similar performance to hardware-based solutions, such as the “small spaces” work on the Pentium [34]. This supports our argument that user-level sandboxing can be implemented efficiently, as a way to support extensibility (assuming hardware restrictions on the working set sizes of logical protection domains).

Also by experiment, we compare various techniques to access sandboxed service extensions. On a Pentium 4 processor, we can safely switch from the kernel to a sandboxed extension function in 11000 cycles, compared to 46000 cycles if we invoke a user-level extension in a private address space that is not currently active. While our approach uses SYSEXIT and SYSENTER machine instructions to quickly switch from the kernel protection domain to user-level, a more portable approach is possible, by establishing an “activation record” on the kernel stack of a running process and then returning to user-level. As a result, we see no reason why our user-level sandboxing technique cannot easily be ported to a number of COTS systems with only minimal kernel modifications.

Future work involves a thorough study of the costs of type-safe languages and cryptographic methods for enforcing trust on sandboxed extension code. Many have argued that hardware methods such as segmentation avoid such costs, but hardware support for fine-grained protection domains (less than a page in size) is not common on many processors. Additionally, we are considering the implementation of sandboxed virtual machines and extensible services for use in a distributed system for processing sensor streams. The idea is to enable low-overhead end-host processing of networked data.

References

- [1] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.
- [2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Ficzynski, and B. E. Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.
- [3] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting gray-box knowledge of buffer-cache management. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, California, June 2002.
- [4] S. Chandra and A. Vahdat. Application-specific network management for energy-aware streaming of popular multimedia format. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, California, June 2002.

- [5] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level resource-constrained sandboxing. In *Proceedings of the 4th USENIX Windows Systems Symposium, 2000*. Seattle, WA, Seattle, WA, 2000.
- [6] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Symposium on Operating Systems Principles*, pages 140–153, 1999.
- [7] D. Clark. The structuring of systems using upcalls. In *Proceedings of Tenth ACM Symposium on Operating Systems Principles*, pages 171–180. ACM, December 1985.
- [8] M. Clarke and G. Coulson. An architecture for dynamically extensible operating systems. In *Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDs'98)*, Annapolis, MD, USA, May 1998.
- [9] Cyclone: <http://www.research.att.com/projects/cyclone/>.
- [10] P. Druschel, V. S. Pai, and W. Zwaenepoel. Extensible kernels are leading os research astray. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, Cape Cod, Massachusetts, 1997.
- [11] Y. Endo, J. Gwertzman, M. Seltzer, C. Small, K. A. Smith, and D. Tang. VINO: The 1994 fall harvest. Technical report, Harvard Computer Center for Research in Computing Technology, 1994.
- [12] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th Symposium on Operating System Principles*, December 1995.
- [13] M. A. et al. Mach: A new kernel foundation for UNIX development. In *Summer USENIX Conference*, pages 93–112, Atlanta, GA, USA, July 1986.
- [14] <http://www.cs.arizona.edu/people/bridges/os/extensible.html>.
- [15] M. E. Fiuczynski and B. N. Bershad. An extensible protocol architecture for application-specific networking. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, January 1996.
- [16] D. Ghormley, S. Rodrigues, D. Petrou, and T. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proceedings of the 1998 USENIX Annual Technical Conference*, June 1998.
- [17] D. P. Ghormley, S. H. Rodrigues, D. Petrou, and T. E. Anderson. Interposition as an operating system extension mechanism. Technical Report CSD-96-920, University of California, Berkeley, September 1997.
- [18] R. P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974.
- [19] G. Gopalakrishnan and G. Parulkar. Efficient user space protocol implementations with QoS guarantees using real-time upcalls. *IEEE/ACM transactions on Networking*, 6(4):374–388, 1998.
- [20] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. Ther performance of μ -kernel-based systems. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*. ACM, October 1997.
- [21] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volumes 1, 2 and 3*.
- [22] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, California, June 2002.
- [23] M. 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.
- [24] P. Joubert, R. King, R. Neves, M. Russinovich, and J. M. Tracey. High-performance memory-based web servers: Kernel and user-space performance. In *Proceedings of the USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.
- [25] J. Lemon. Kqueue - a generic and scalable event notification facility. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2001.
- [26] J. Liedtke. On μ -kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, December 1995.
- [27] G. Morrisett, K. Cray, N. Glew, D. Grossman, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1999.
- [28] G. Morrisett, D. Walker, K. Cray, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.
- [29] G. C. Necula and P. Lee. Safe kernel extensions without runtime checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 28–31, 1996. Seattle, WA, pages 229–243, Berkeley, CA, USA, 1996.
- [30] T. A. R. Wahbe, S. Lucco and S. Graham. Software-based fault isolation. In *Proceedings of the 14th SOSP*, Asheville, NC, USA, December 1993.
- [31] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.
- [32] C. Small and M. I. Seltzer. A comparison of OS extension technologies. In *USENIX Annual Technical Conference*, pages 41–54, 1996.
- [33] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMWare workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.
- [34] V. Uhlig, U. Dannowski, E. Skoglund, A. Haeberlen, and G. Heiser. Performance of address-space multiplexing on the Pentium. Technical Report 2002-1, University of Karlsruhe, Germany, 2002.
- [35] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. Ashs: Application-specific handlers for high-performance messaging. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM '96)*, 1996.
- [36] R. West and J. Gloudon. 'QoS safe' kernel extensions for real-time resource management. In *the 14th EuroMicro International Conference on Real-Time Systems*, June 2002.