17th ACM Symposium on Operating Systems Principles (SOSP '99),
Published as *Operating Systems Review*, **34**(5):140–153, Dec. 1999

# Integrating segmentation and paging protection for safe, efficient and transparent software extensions

Tzi-cker Chiueh   Ganesh Venkitachalam   Prashant Pradhan

Computer Science Department
State University of New York at Stony Brook

*chiueh, ganesh, prashant@cs.sunysb.edu*

## Abstract

*The trend towards extensible software architectures and component-based software development demands safe, efficient, and easy-to-use extension mechanisms to enforce protection boundaries among software modules residing in the same address space. This paper describes the design, implementation, and evaluation of a novel intra-address space protection mechanism called* Palladium, *which exploits the segmentation and paging hardware in the Intel X86 architecture and efficiently supports safe kernel-level and user-level extensions in a way that is largely transparent to programmers and existing programming tools. Based on the considerations on ease of extension programming and systems implementation complexity,* Palladium *uses different approaches to support user-level and kernel-level extension mechanisms. To demonstrate the effectiveness of the* Palladium *architecture, we built a Web server that exploits the user-level extension mechanism to invoke CGI scripts as local function calls in a safe way, and we constructed a compiled network packet filter that exploits the kernel-level extension mechanism to run packet-filtering binaries safely inside the kernel at native speed. The current* Palladium *prototype implementation demonstrates that a protected procedure call and return costs 142 CPU cycles on a Pentium 200MHz machine running Linux.*

## 1 Introduction

Two emerging trends in applications software development call for operating systems support for establishing protection boundaries among program modules that execute in the same address space. First, the notion of *dynamic extensibility* has prevailed in almost every major category of software systems, including extensible database systems [26], to which third-party data blades can be added to perform type-specific data processing, extensible operating systems [6, 15, 23], which support application-specific resource management policies, programmable active network devices [1, 27] that allow protocol code running on network devices to be tailored to individual applications, and user-level applications that dynamically integrate third-party modules to augment the applications' core functionalities such as Adobe's Premiere [12] and Apache Web Server [2]. A distinct feature of extensible software systems is support of *live* addition and removal of software modules into and from a running program. Because the host program and the extension software modules share the same address space, an effective and efficient mechanism to protect the core of the running host program from dynamically inserted extension modules is crucial to the long-term viability of extensible software architecture. Second, component-based software development (CBSD) [18] is emerging as the dominant software development methodology because it significantly improves software productivity by encouraging modularity and re-usability. As software components produced by multiple vendors are used to construct complete applications, a proper level of protection among software components is essential to address the key challenge of the CBSD methodology: prevention of interference among independently developed components and the resulting loss of system robustness. Appropriate inter-component isolation makes it is easier to quarantine buggy components and pinpoint the cause of application malfunctioning.

Although a number of approaches have been proposed to provide intra-address space protection, such as software fault isolation [29], type-safe languages [6], interpretive languages [17], and proof-carrying code [19], none satisfies all the design goals of an ideal intra-address space protection mechanism: safety from corrupting extension modules, low run-time overhead, and programming simplicity. The commonality among all the above approaches is the use of software-only techniques to create protection domains within an address space. The implicit assumption of these approaches is that hardware-based protection mechanisms are only applicable to inter address-space protection. In contrast, this paper describes an intra address-space protection mechanism called *Palladium*, which is based on the segment-level and page-level protection hardware in the Intel X86 architecture. *Palladium* is efficient, guarantees the same level of safety as using separate address spaces, and requires only modest efforts in the deployment and development of software extensions. Although the proposed mechanism is geared towards the Intel X86 architecture, the fact that this architecture dominates more than 90% of the world's desktop computer market implies that it can have wide applicability and thus see practical uses.

The basic idea of *Palladium* to protect an extensible application from its extensions is to put the core program and its extensions in disjoint segments that belong to the same address space but are at different protection levels. Because software extensions are put at a less privileged protection level than the core program, they cannot access the core program's address space without proper authorization. This approach is possible because the Intel X86 architecture supports variable-length segments and multiple segment protection levels. Unfortunately, this approach significantly complicates the interfaces between extensible applications and their extensions because cross-segment references require changes to the underlying pointers and thus put additional burdens on application programmers and/or compiler writers. While the requirement of changing inter-segment pointers is acceptable for kernel extensions, it is considered too drastic for user-level extensions. As a result, we developed a separate protection mechanism that exploits the page-level and segment-level protection hardware features of X86 architecture to support user-level extensions without requiring pointer modifications. This second mechanism significantly improves the transparency of extensions programming compared to the segment-only approach.

The rest of this paper is organized as follows. Section 2 reviews previous works on supporting intra-address space protection. Section 3 details the virtual memory support from the Intel X86 architecture. In Section 4, we describe *Palladium*'s protection and protected control transfer mechanisms to support kernel and user-level extensions. Section 5 presents a comprehensive performance study of *Palladium* based on measurements from a user-level extensible application for fast CGI script invocation, and a kernel-level extensible application for packet filtering. Section 6 concludes this paper with a summary of main results and an outline of the on-going work.

## 2 Related work

Previous approaches to fast communications between protection domains attempt to either establish protection boundaries within an address space or reduce the IPC overhead between address spaces. Most of them focused mainly on kernel-level extensions but not on user-level extensions. In this section, we review important ideas from these efforts and conclude with a comparison between them and *Palladium*.

### 2.1 Providing protection within an address space

Multics [4, 11] pioneered the use of segmentation and ring-like protection hardware, which is available in GE-645 machines, in virtual memory architecture. Segments are visible to application programmers and are used to host code, stack, data, and even files and directories. Data sharing within a process or among processes is controlled through segment-level protection checks. Unlike the X86 architecture, the paging hardware in GE-645 does not support page-level protection. Paging in GE-645 is mainly for performance optimization rather than for protection. *Palladium*'s kernel-level extension mechanism is similar to Multics, but its user-level extension mechanism is quite different. *Palladium* exploits both page-level and segment-level protection checks to hide segmentation from application programmers and existing programming tools.

HP's PA-RISC architecture [21] provides the most comprehensive protection and security hardware support among modern RISC machines. Like X86, PA-RISC has 4 privilege levels. Similar to segments, PA-RISC supports the notion of multiple *protection identifiers* per process. A page with a given *access identifier* is only accessible to a process with the matching protection identifier. By associating different sets of protection identifiers with different code modules in the same process, PA-RISC can support multiple protection domains within a single address space. However, except a brief mention in Brevix [30], no general OS extension mechanisms built on top of this architectural feature have been reported in the literature. Opal, which is a single-address-space OS [7], also used a similar protection-domain identifier idea to enforce protection boundaries within an address space.

One software-only approach to provide intra-address space protection is to interpret rather than execute an extension. Protection can be guaranteed only if the interpreter itself is correct. An example of this approach is Java-based systems, where the language itself is type-safe and does not allow arbitrary pointer accesses [10], and run-time interpretation of Java programs can perform additional checks to detect bugs such as those that cause denial of service [8]. For example, the HotJava Browser can be extended with applets written in Java [24]. Another example is the Berkeley Packet Filter, in which the kernel interprets filtering rules submitted by the applications [17]. There are two problems with the interpretation approach. First, the safety/security offered by this approach is only as strong as the interpreter implementation. For example, there have been a number of security-related bugs discovered in Java virtual machine implementations. The problem is that software systems remain difficult to verify. The second problem is that Java applications are still less efficient than their C counterparts, either because of run-time interpretation [24] or because of additional type checking and garbage collection cost when a just-in-time compiler is used.

In software fault isolation (SFI), an extension is *sandboxed* so that any memory accesses it makes are guaranteed to fall within the memory region allocated to the extension [29, 25]. Additional instructions are inserted to the extension's binaries to force memory accesses to fall into the extension's allocated region. The protection offered to applications can be write protection (only writes made by the extension are forced), or read-write protection (all memory accesses are forced). VINO [23] is an extensible kernel that uses SFI. The overhead imposed by SFI ranges from under 1% to 220% of the execution time of an unprotected extension running in the same address space.

Another way to protect an extensible application from its extensions is to write the extensions in a type-safe language, such as Modula-3. Because of the language restrictions, the extension cannot access the application memory and corrupt the core program. This is the approach taken by SPIN [6]. The SPIN OS kernel itself is written in Modula-3, and it is possible to extend the kernel at the granularity of individual functions by dynamically linking code written in Modula-3 into the kernel. Protection is ensured by both compile-time and run-time checking performed by the language compiler and run-time system. The difference between this approach and SFI is that the application depends on the Modula-3 compiler to generate code for run-time checking. A buggy compiler can actually allow extension code to corrupt the application, and at least once, such an incident has occurred [24]. The overhead percentage of this approach depends on the types of operations that extensions perform, and has been found to range from 10% to 150% of the same code written in C.

The protected shared libraries project [3] attempted to build sensitive systems services as user-level libraries, rather than into the kernel. The implementation on AIX 3.2.5 still required context switching for protection-domain crossing, and thus suffered from a much higher performance overhead compared to *Palladium*.

## 2.2 Reducing IPC overhead

Lightweight RPC (LRPC) [5] reduces the overhead associated with making an RPC call to a server process executing in the local machine by optimizing data copying and network-related processing operations. In LRPC, a server module registers itself with the kernel and exports a set of procedures to the kernel by creating a Procedure Descriptor List. Each Procedure Descriptor in the list will have an associated argument stack. The client and server share the argument stack when a procedure in the server is invoked. This eliminates copying data multiple times. Since the client, the kernel, and the server have foreknowledge that the particular RPC call is to a server running on the local machine, argument marshaling overhead can be eliminated and simple byte copying can be used. Further, the server and client stubs are directly invoked by the kernel with a simple upcall. The result is that LRPC performs up to four times faster than conventional RPC. On a C-VAX Firefly machine, LRPC requires 125 $\mu$secs for a Null function call to complete, compared to 464 $\mu$secs for a more conventional RPC call. Note that two context switches and four protection domain crossings must still be performed by the LRPC mechanism for a request-reply transaction.

The L4 micro-kernel [16] achieved extremely fast IPC performance by sharing page tables between multiple processes. On the Intel Pentium Processor, the kernel ensures that processes are protected from each other by reloading the segment registers on a context switch. Thus a page table switch and the associated TLB flush is avoided when possible. In an L4 micro-kernel running on an Intel Pentium 166MHz Processor, an IPC request-reply requires a minimum of 242 cycles, or 1.46 $\mu$secs in the ideal case. However, four protection domain crossings are needed for a request-reply transaction. Moreover, if the sum of the virtual address spaces covered by the segments of active processes exceeds the 4-GByte virtual address space available in the processor, the kernel either has to prevent further processes being spawned or incurs the overhead of a page table switch. Explicit data copying is still inevitable in L4 for processes to share data. Another single-address-space OS, Mungi [13], is built on L4's fast IPC to support sparse capabilities and fast protected procedure calls on MIPS-R4600 64-bit microprocessor.

## 2.3 Comparison

Software-only approaches to support intra-address space protection that are based on SFI, interpretation, or type-safe languages requiring dynamic typing checks, incur an overhead that is approximately proportional to the amount of extension code executed. In addition, the protection guarantees provided by software-only approaches is only correct if the implementation of the compiler, the interpreter, or the binary patching tool is bug-free. Past experiences indicate that this need not always be the case.

Hardware-based protection mechanisms do not incur per-instruction overhead beyond the processor-level performance cost. The cost of invoking an extension is typically a one-time cost associated with each protection-domain crossing. Hardware design is also more likely to be tested extensively or verified formally, and thus is less buggy compared to software at the same level of product maturity. In addition, hardware-based approaches do not require substantial changes to existing programming practices, and thus greatly simplify extension programming. It is not necessary for developers to learn new programming languages or to drastically change current programming styles to compose safe extension modules. Until now, very few extension mechanisms have exploited segmentation hardware support [16] to support multiple protection domains *within* an address space, especially at both the user and kernel levels. To the best of our knowledge, *Palladium* is one of the first, if not the first such successful attempts.

## 3 Protection hardware features in Intel X86 architecture

## 3.1 Protection checks

Intel X86 architecture's virtual memory hardware supports both variable-length segments and fixed-sized pages, as shown in Figure 1. A virtual address consists of a 16-bit

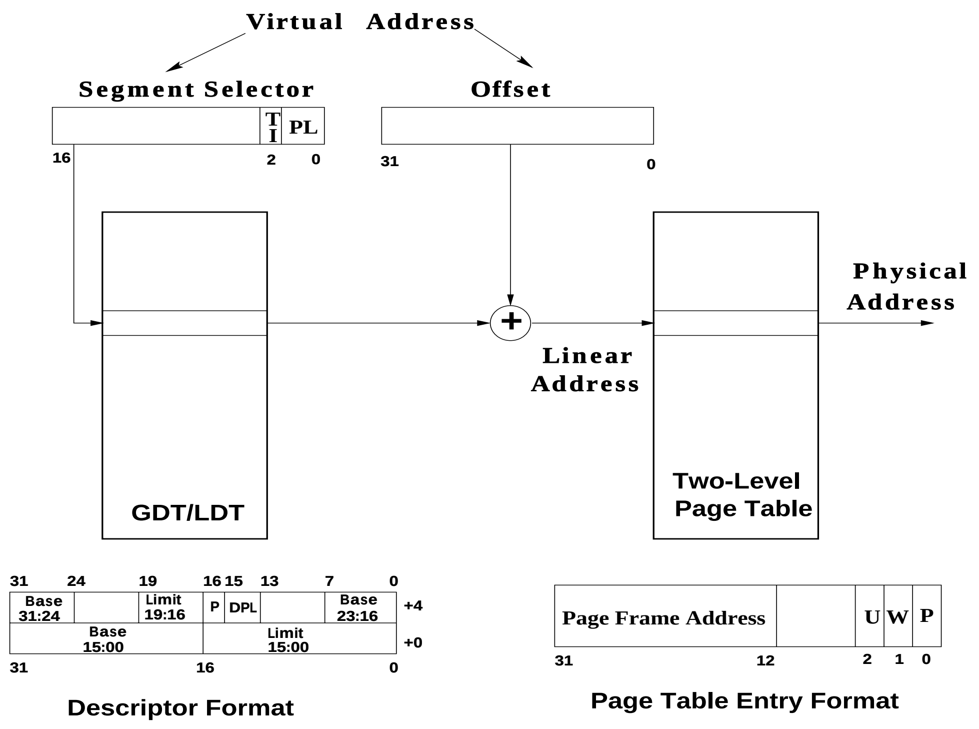


**Figure 1**. The virtual memory architecture of Intel X86 architecture, which provides both segment-level and page-level protection checks, and supports variable-length segments as well as a 4-level protection ring. For each memory access, the hardware performs checks for segment limit violation, segment-level and page-level protection violation, and read/write permission. To speed up the translation and protection check process, modern X86-based processors include a Translation Lookaside Buffer (TLB), which is automatically flushed on task switch.

*segment selector*, which is in one of the on-chip segment registers, and a 32-bit *offset*. which is given by `EIP` register for instruction references, `ESP` register for stack operations, or other registers/operands in the case of data references. The segment selector is an index into the *Global Descriptor Table* (GDT) or the current process's *Local Descriptor Table* (LDT). The choice between GDT and LDT is determined by a TI bit in the segment selector. The GDT or LDT entry indexed by the segment selector contains a *segment descriptor*, which, among other things, includes the start and limit addresses of the segment, the segment's descriptor privilege level (DPL), and R/W read/write protection bits. The 32-bit offset is added to the given segment's start address to form a 32-bit *linear address*. The most significant 20 bits of a linear address are a virtual memory page number and are used to index into a two-level page table to identify the corresponding physical page's base address, to which the remaining 12 bits are added to form the final physical address. The page size is 4 KBytes.

Each segment can be in one of four possible segment privilege levels (SPL), which is specified in the DPL field of the segment's descriptor. Each virtual page can be in one of two possible page privilege levels (PPL). SPL 0 is the most privileged level and SPL 3 is the least privileged level. Similarly, PPL 0 is more privileged than PPL 1. By default, pages that belong to segments at SPL between 0 to 2 are mapped to PPL 0 while pages that belong to segments at SPL 3 are mapped to PPL 1. Therefore, code segments at SPL 3 do not have the privilege to access pages at PPL 0. The segment privilege level of the currently executing code is stored in the last two bits of the Code Segment register.

Intel X86 architecture provides protection checks at both segment and page levels. After a linear address is formed, the hardware checks whether it is within the corresponding segment's limit as specified in the segment descriptor. Program execution based on code residing at a less privileged level, i.e., with a higher SPL, cannot access data segments or jump to code segments that are at a more privileged level, i.e., with a lower SPL. At the page level, the protection hardware ensures that programs executing at SPL 3 cannot access a page marked as PPL 0 and programs executing at SPL 0 to 2 can access all pages. With segmentation checks, each segment can form an independent protection domain if segments are disjoint from one another.

CPU control registers that are related to protection, including the base address registers for LDT and GDT, and the registers that point to the starting address of the current process's *Task State Segment* (TSS) and page table, TR and CR3 can only be modified by code running at SPL 0. The TSS of a process holds, among other things, the base physical address of the process's page table. On a task switch, the hardware automatically loads CR3 using the information from TSS, and flushes the TLB. Finally, a code segment cannot lower its SPL without invoking a kernel service via Interrupt gates.

## 3.2 Control transfer among protection domains

While the protection mechanisms described in the previous subsection successfully confine the instruction and data accesses of a code segment to domains at the same or less privileged levels, there are legitimate needs for less privileged programs to access data or instructions at more privileged levels. One of such mechanisms provided by Intel X86 architecture is the *call gate*. A call gate is described by a 8-byte segment descriptor entry in the GDT or LDT. To make an inter-segment or inter-privilege-level procedure call, the `lcall` instruction is used in conjunction with a call gate ID. Each call gate entry also contains a descriptor privilege Level that specifies the minimum privilege level required to access this call gate and an entry point to which the control is first transferred in every invocation of this call gate. Because call gates themselves reside in the GDT/LDT, and thus are modifiable only by code running at SPL 0, normal user-level code cannot change them to gain unauthorized accesses.

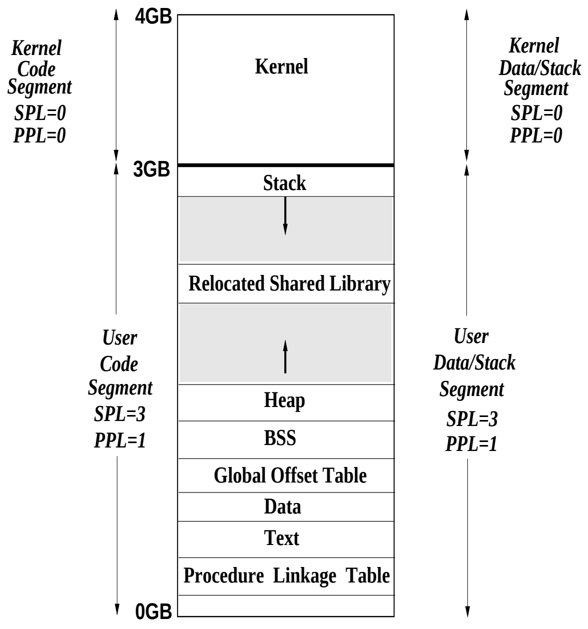
To prevent corruption through stacks in inter-privilege-level procedure calls, each privilege level has its own stack. Stack switching is required for procedure calls that cross privilege levels. Each process's TSS has three stack pointers, one for SPL 0, 1, and 2, and each consists of a segment selector and an offset. TSS does not keep a separate stack pointer for SPL 3, because X86 architecture does *not* allow a more privileged routine to call a less privileged routine. Note that there is still a stack specifically for SPL 3, but SPL 3's stack pointer does not have to be explicitly stored in the TSS.

## 4 Intra-address space protection

## 4.1 Extension programming model

*Palladium* supports safe and dynamic extensions at both user and kernel levels and assumes the following extension programming model:

- A core program, the kernel or an extensible application, is protected from dynamically-linked extension modules but not vice versa. Among extension modules, the protection is only for safety but not for security.
- Extensions are protected function calls, which are single-threaded and always run to completion. The extensions of all existing extensible operating systems [23, 6, 15] are also based on this function call model.
- To avoid data copying, extensions and the core program can share data through specific data areas that could be chosen at run time.
- User extensions cannot make arbitrary system calls without going through hosting applications, and kernel extensions can access only certain core kernel services as determined by the kernel.



4GB
3GB
0GB
Kernel Code Segment SPL=0 PPL=0
Kernel Data/Stack Segment SPL=0 PPL=0
User Code Segment SPL=3 PPL=1
User Data/Stack Segment SPL=3 PPL=1
Kernel
Stack
Relocated Shared Library
Heap
BSS
Global Offset Table
Data
Text
Procedure Linkage Table

**Figure 2**. The layout of a Linux process's virtual address space. The Procedure Linkage Table and Global Offset Table used in dynamic loading/linking. Shared libraries are memory mapped to the middle of the unused region between Heap and Stack. The shaded areas are free regions.

## 4.2 Virtual address space structure in Linux

The current prototype implementation of *Palladium* is based on Linux 2.0.34. In Linux, the 4GByte virtual address space (VAS) is arranged as follows. The User Code segment spans 0 to 3GByte. The User Data/Stack segment also spans 0 to 3GByte. Both segments are accessible to user-level processes and are set at SPL 3 and PPL 1. The Kernel Code and Data/Stack segments both span 3GByte to 4GByte, and are set at SPL 0 and thus protected from user processes. Kernel segments are always present in the GDT and thus are a part of *every* running user process, but they are only accessible through Interrupt gates. In summary, a Linux process's VAS has 4 segments: two user segments spanning 0 to 3GByte and two kernel segments spanning 3GByte to 4GByte. The protection of kernel segments from user segments are through both segment limit and SPL checks.

Figure 2 shows the layout of the virtual address space of a Linux process. The user code is loaded at a starting address a little bit greater than 0, thus leaving a hole at the bottom. This hole is to map the code/data in `ld.so` that performs relocation. Text is the code region, Data is the initialized data region and BSS is the uninitialized data region. Heap grows towards the Kernel segment whereas Stack grows away from the Kernel segment. Global Offset Table and Procedure Linkage Table are used to support dynamically linking/loading. The unused areas between Stack and Heap are shown as shaded zones in Figure 2. Files can be memory mapped into any free area in the 0-3GByte range. For example, shared libraries are usually mapped into the middle of the 0-3GByte range when they are loaded.

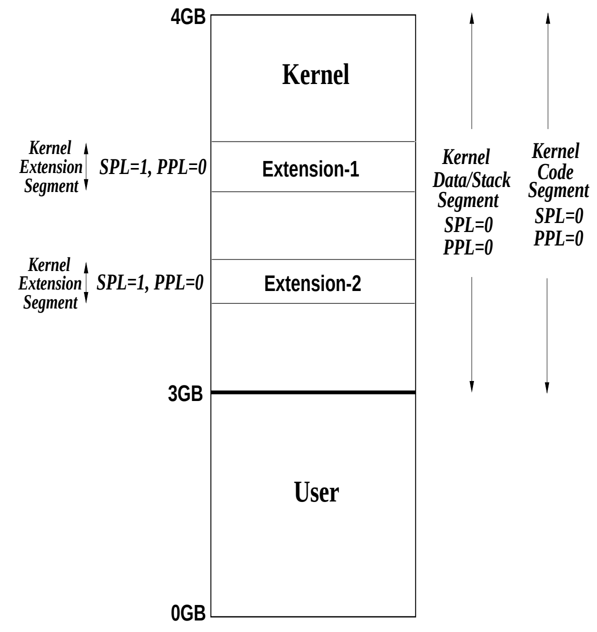


**Figure 3**. The layout of the kernel portion of a *Palladium* process's virtual address space. One or multiple extension segments, in this case 2, can be loaded into the kernel address space, i.e., 3-4GByte, and they are put at SPL 1 and PPL 0.

## 4.3 Safe kernel extension mechanism based on segment-level protection

Linux can load modules into the kernel dynamically using the `insmod` utility. A loadable kernel module, once loaded, is effectively part of the kernel in the sense that it can access anything accessible to the kernel. The goal of *Palladium*'s safe kernel extension mechanism is to prevent buggy kernel extension modules from corrupting the kernel address space and crashing the entire system. The basic idea of protecting the kernel from its extension modules is simple: load each extension module into a separate and less privileged segment that falls completely within the kernel address space, as shown in Figure 3. Specifically, a special *extension segment* that spans a subrange of the kernel address space, i.e., between 3GByte and 4GByte, and has its SPL set at 1, is created to hold extension modules. The kernel can still access everything in the extension segment, but the extension module is confined to its own segment because any attempts to access the portion of the kernel address space that is outside the extension segment will cause either segment limit check or SPL check to fail.

Figure 4 illustrates the interaction between a user process, the kernel, and a kernel extension. A user process requests a specific kernel service by calling an *interrupt gate* (Step 1), which first performs necessary checks, switches to a per-process kernel stack and saves the code/stack pointers for the user process, and jumps to the corresponding kernel routine by indexing into the System Call Table (Step 2 and 3). After the kernel service is completed, the user process's state is restored and the control returns to the user process (Step 10).

*Palladium* loads an untrusted kernel extension into an extension segment, including its code, data, and stack structures. Because the extension segment's SPL is 1, it will never be able to corrupt the part of the kernel address space outside the extension segment. There is only one stack for each extension segment; that stack is allocated when the first module is loaded into that extension segment. One or more modules can be loaded into an extension segment. Modules loaded into the same extension segment can share a single stack because *Palladium* assumes that they will not run concurrently. *Palladium* does not provide protection among software modules loaded into the same extension segment. However, inter-module protection could be easily supported by creating one extension segment per module. Modules that share an extension segment can freely share data among themselves without cross-segment data movement.

Whenever a new extension is loaded into the kernel, it registers with the kernel one or multiple function pointers as extension service entry points. The kernel keeps an Extension Function Table for these functions and invokes new extension services as needed. *Palladium* first checks for the existence of a given extension by name (Step 4). If the required extension service has not yet been instantiated, no action is taken; otherwise the corresponding service is invoked (Step 5 and 9).

Although extension modules are confined to the extension segment, they may access kernel routines and states through a pre-defined interface that resembles a conventional user-kernel system-call interface (Step 6, 7, and 8), which in the current implementation is designed specifically for a programmable network router [22]. The kernel service function called by an extension module executes in the kernel stack of the user process that triggers the kernel extension. If the kernel does not act on any user process's behalf when invoking a kernel extension, such kernel service functions execute in the stack of the *idle* process. The execution of a kernel extension is expected to be entirely self-contained, i.e., without any kernel service invocation. For example, packet filters, new protocol stacks, and new device drivers have been shown to be implementable in user space, where besides parameter passing, interaction with the kernel is only needed during the initial set-up and final result-passing phases, but not during the execution of the main body of extensions. However, to facilitate and simplify kernel extension programming, *Palladium* chooses to expose a set of core kernel services without compromising safety/security.

In addition to synchronous function calls, *Palladium* also supports a primitive form of *asynchronous* extensions. In this case, the kernel puts a request into the target extension module's request queue, marks the module *busy*, and returns. When extensions that are busy are scheduled for execution, they pick up a request from their queue and run that request to the completion before servicing the next. Asynchronous extensions are used to support extension functions that are not re-entrant but may be called independently from multiple points in the kernel while the previous invocation is still in progress. For example, an incoming packet can be queued for the asynchronous service of protocol-specific packet filtering, if the CPU is busy with other high-priority tasks on packet arrival. Because each extension segment has its own

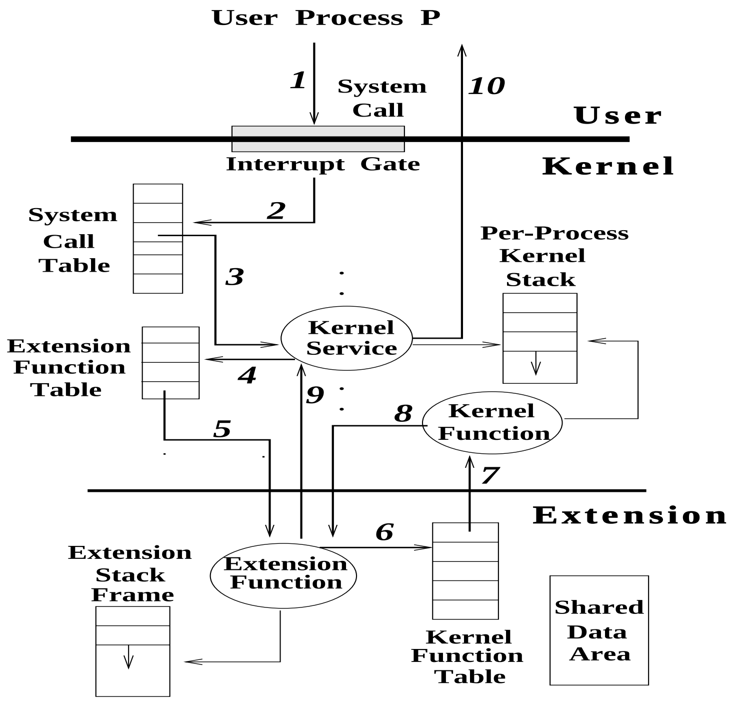


**Figure 4**. Interactions between a user process, the kernel, and a kernel extension. A simple system call that does not involve kernel extensions takes the path **1-2-3-10**. A system call that requires the service of a self-contained kernel extension takes the path **1-2-3-4-5-9-10**. Finally, a system call that requires the service of a kernel extension, which in turns requires some kernel service, takes the path **1-2-3-4-5-6-7-8-9-10**

stack, both synchronous and asynchronous extensions execute in the stack associated with their extension segments.

To facilitate data sharing and reduce data copying between the kernel and extension modules, an extension can allocate a shared data area inside its extension segment (shown in Figure 4), to which the kernel can pass arguments into and out of extension functions. The shared area is given a well-known symbol, which the kernel checks for existence at run time. This shared area is read/write accessible to both the kernel and extension modules and is meant to hold non-sensitive data during extension processing, e.g., the headers of network packets that need to be examined by both the kernel and its extensions.

*Palladium's* kernel extensions are written as kernel modules and are loaded into the kernel using a modified version of `insmod`. Extension programming is identical to kernel module programming, except that they can build on the set of core kernel services exposed to kernel extensions.

## 4.4 Safe user-level extensions

### *4.4.1 Combining paging and segmentation protection*

A user-level process in Linux can also dynamically load an extension module into its address space using `dlopen`, `dlsym` and `dlclose`. Similar protection issues arise between an extensible application, such as an extensible database management system, and its extension modules, such as type-specific access methods. Although the segmentation-based kernel extension mechanism described in the previous subsection could be applied to supporting safe user-level extensions in theory, the following considerations motivate us to develop a separate protection mechanism for user-level extensions.

First, directly applying the segmentation-based approach to user-level extensions makes it difficult to share code or data between an extensible application and its extensions. Because the extended program and the extensions have different base addresses, pointers need to be swizzled before being passed among segments. In a similar vein, the Linux kernel interprets the pointer arguments passed through system calls with respect to the base address 0. If extensions are allowed to make system calls directly, the Linux kernel has to identify the calling code segment's base address and adjusts the pointer arguments accordingly, for every system call.

Secondly, the relocation routines in the current dynamic library package need to be modified to load extensions to an extension segment with a different base address than 0. Because `gcc` and `ld` assume a linear virtual address space architecture, they are not designed for segments.

Finally and most importantly, extensions cannot share standard libraries with the extended program, because library routines such as `libc` are in the application segment but outside any extension segment. Putting `libc` inside an extension segment is not a solution to this problem because there may be multiple extension segments. Actually, this scheme leads to a potential security vulnerability since extensions may corrupt the extended application by damaging the data areas used by `libc` functions that have internal buffers such as `fprintf`. Linking each extension module statically with all the library functions it needs is another possibility. Unfortunately, this approach not only wastes memory, but is also incorrect, because a buffering and thus stateful library function may have multiple copies residing in the same address space simultaneously. Note that the `libc` problem does not exist in the context of kernel extensions.

Instead of pure segmentation, we chose an approach that uses both paging and segmentation protection hardware to support safe user-level extensions, as shown in Figure 5. An application process starts at SPL 3 by default and, if it is meant to be extensible, it then promotes itself to SPL 2 through an `init_PL` system call which sets the PPL of all the process's writable pages to 0 and creates an extension segment that is at SPL 3 and spans 0 to 3GByte. Finally, the PPL of any pages that the extended application wants to expose to user-level extensions, such as code in shared libraries or data regions shared between the application and extension, is set to 1. This data sharing mechanism dictates that the size of the shared data area be a multiple of the page size. It may also lead to additional data copying unless the shared data is carefully placed when they are generated.

Because the application and extension segments have the same base address, a user-level extension can access anything in the 0-3GByte address range at the *segment* level, i.e., the segment-level protection checks will go through. However, at the *page* level, an extension cannot access those pages that the application chooses to hide and therefore are at PPL 0, because the paging hardware prevents SPL 3 code segments from accessing PPL 0 pages. Therefore, extensions can only access their own code, data, and stack, as well as shared libraries and data regions exposed by the extensible application. On the other hand, although the pages in the kernel address space (3-4GByte) and the user address space (0-3GByte) are all at PPL 0, the extensible application cannot access the kernel address space because of segment-level protection. In summary, segment-level check ensures that the kernel is protected from the extensible applications, and page-level check protects the extensible applications from their extensions, exactly the protection guarantees we are looking for!

Because the extended application's segment and the extension segments cover exactly the same virtual address space range, all the problems associated with the segmentation approach disappear. The relocation mechanism in `dlsym` is directly applicable without any modification. Data and function pointers can be passed among the kernel, the extended application, and extensions without swizzling. Extensions can call directly non-buffering `libc` routines such as `strcpy`, because their pages are set at PPL 1. The data areas of `libc` are at PPL 0. Therefore, extensions cannot call buffering library routines such as `fprintf` directly. Instead, *Palladium* allows applications to expose *application services* to extensions, much as the kernel exposes core kernel services to to kernel extensions as shown in Figure 4. Only buffering library functions in `libc` are required to be encapsulated as applications services, which extensions can call but cannot corrupt. Unlike in the segmentation-only approach, extensions can call non-buffering library functions without crossing protection domains.

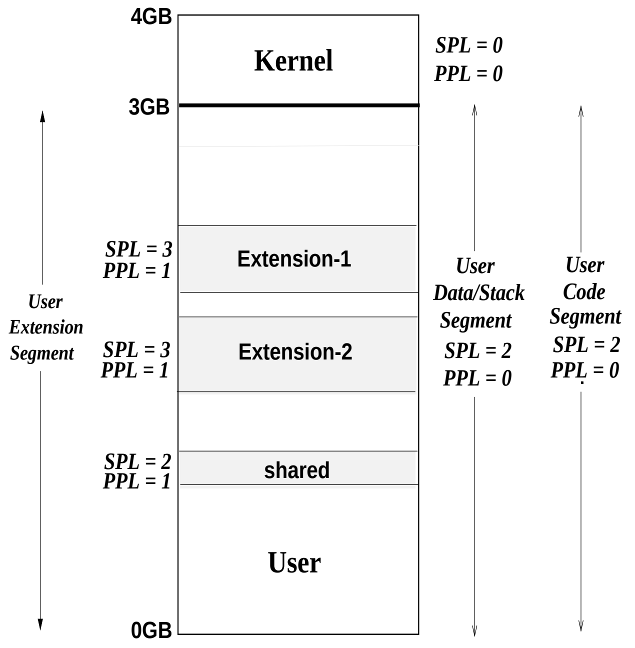


**Figure 5**. The layout of the user portion of a *Palladium* process's virtual address space. One or multiple extension segments, in this case 2, can be loaded into the user address space (0-3GByte) and put at SPL 3, and the pages therein are at PPL 1. The extended application itself is at SPL 2, and its pages at PPL 0, except those that are to be shared with extensions, which are at PPL 1.

### 4.4.2 Programming interface

To use *Palladium*'s user-level extension mechanism, extensible applications are required to use a safe version of the dynamic loading package, i.e., `seg_dlopen`, `seg_dlsym`, and `seg_dlclose`, to load, access, and close shared libraries. However, `seg_dlsym` should be used only for acquiring function pointers. To resolve pointers to data structures inside an extension segment, `dlsym` should be used instead. In addition, the extensible application should call the `init_PL` function in the beginning of the program to promote itself to SPL 2 and mark all its writable pages as PPL 0. To expose shared pages to extensions, the application can use the `set_range` system call to mark those pages as PPL 1. To expose an application service that user-level extensions could use, the application uses the `set_call_gate` system call to set up a call gate with a pointer to the corresponding application service function.

Programming user-level extensions is identical to developing a user-level library routine, except that `xmalloc` instead of `malloc` should be used to ensure that it's the extension segment's heap that is being allocated. *Palladium*'s extensions are compiled with `gcc`, just like conventional shared libraries. Calling an extension function from an application and returning from a called extension back to the calling application follow exactly the standard C syntax, although applications and extensions reside at different privilege levels.

Each dynamically-linked function has a corresponding Global Offset Table entry (GOT) and Procedure Linkage Table (PLT) entry. When a dynamically linked function is called, control first goes to the corresponding PLT entry, which contains a `jmp` instruction that jumps where the associated GOT entry points. The first time a dynamically-linked function is invoked, its GOT entry points to the relocation function in `ld.so`, which loads the function, performs necessary relocation, and modifies the GOT entry to point where the function is actually loaded so that all subsequent invocations would transfer control directly to the function. Because extensions need to access GOT to invoke shared libraries, the GOT should be marked as PPL 1 and should be put in a separate page to protect its neighboring regions, such as BSS. `Gcc` uses an internal linker script to define the placement of various sections of the program image, such as Text and Data. *Palladium* requires applications to be compiled with a specific `gcc` linker script that ensures that the GOT is aligned on a page boundary. To protect the GOT from being corrupted by extensions, *Palladium* marks the GOT page as read-only by requiring that all modifications to the GOT be made in the beginning of program execution. This means that when the application and its extensions are loaded, the symbols within them should be resolved eagerly, not lazily.

## 4.5 Implementation

### *4.5.1 Control transfer*

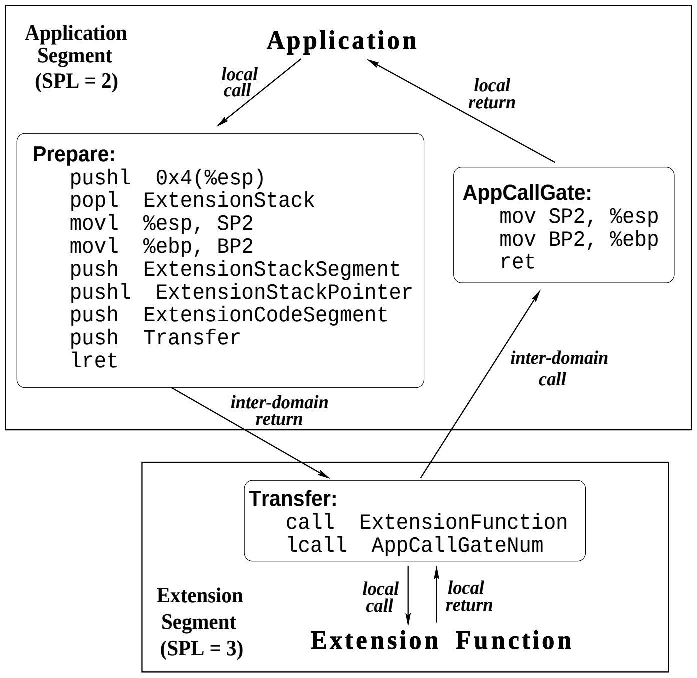
*Palladium* has to solve two problems related to transferring control between extended programs and their extensions. First, the X86 architecture assumes that the control between protection domains always starts from a less privileged level to a more privileged level and back, as a standard system call does. That is, a more privileged code segment can only *return* to a less privileged code segment that *called* on its service previously. A more privileged code segment cannot directly *call* a less privileged code segment. However, *Palladium*'s extension model is meant for less privileged modules to extend the functionality of more privileged extended programs (the kernel or extensible applications), so the control transfer is actually initiated by the more privileged core programs. Second, `gcc` and `ld` have no knowledge of segments, and it is essential to keep *Palladium* completely transparent to `gcc` and `ld` to increase its applicability.

The solution to both problems is to add one level of code indirection. Specifically, three code sequences are added to hide the details of inter-domain control transfers and the call/return semantics mismatch between Intel's hardware and *Palladium*'s requirements, as shown in Figure 6. These code sequences basically perform inter-domain control transfer and stack pointers save/restore to twist X86's `lret` and `lcall` instructions to achieve the desired effects. To invoke an extension function, the application first makes a normal function call to an extension-function-specific `Prepare` routine, running at SPL 2, which passes the input argument to the extension stack, saves the application's stack and base pointers, constructs a phantom activation record that corresponds to the target `Transfer` function's stack and code pointers, and finally executes a `lret`. The phantom activation record is set up such that the control would *return* to another extension-function-specific `Transfer` routine, which is at SPL 3, as if this `Transfer` routine called `Prepare` previously. The `Transfer` routine then simply makes a local function call to the target extension function to perform the extension service. When the extension function is completed, it returns to the `Transfer` routine, which makes an inter-domain call via a call gate to an application-specific `AppCallGate` routine, which restores the extensible application's stack and base pointers, and makes a local `ret` to transfer control back to the extended application.

In summary, a logical *call* from a more-privileged to a less-privileged domain is implemented physically via two intra-domain `call`s and an inter-domain `lret` instruction, whereas a logical *return* from a less-privileged to a more-privileged domain is implemented physically as two intra-domain `ret`s and an inter-domain `lcall` instruction. Note that the `Transfer` and `Prepare` routines are specific to each extension function, but `AppCallGate` is per application. When `seg_dlsym` is invoked to resolve a function symbol, it generates the `Transfer` and `Prepare` routines, and returns a pointer to the corresponding `Prepare` function, rather than to the original extension function. Because only function pointers in extension segments need to be "massaged" when they are loaded, data pointers in extension segments can still be resolved by `dlsym`.

Saving and restoring the application's base and stack pointers in `Prepare` and `AppCallGate` is mandatory, because Intel hardware automatically restores the stack pointer of the corresponding SPL from the process's Task State Segment after an *lcall*. However, because the corresponding `Prepare` routine does not save the application stack pointer to the TSS, the stack pointer that the hardware restores after `AppCallGate` is called is *not* the calling application's stack. Consequently, explicit saving and restoring is required. While *Palladium* could have chosen to save the extended application's stack pointers to the TSS so that what the hardware restore is correct, doing so would incur an expensive system call overhead required to access the TSS, and defeats the whole purpose of using segmentation hardware for fast protected extension calls. Instead, *Palladium* saves the stack/base pointers in the application segment, and spends two additional instructions in `AppCallGate` to put them back.

*Palladium* also allows applications to provide application services to user extensions. The control transfer between user extensions and application services is similar to control transfer in standard system call invocations, except the following differences. Unlike system calls, which typically run in per-process kernel stacks, the application service called by an extension executes against the extension segment's own stack rather than against the application segment's stack. This design choice improves transparency because the standard parameter passing mechanism used by `gcc` is directly applicable, including the support for functions with variable numbers of arguments. In addition, no cross-segment data copying is required. The current *Palladium* implementation assumes that extensions take one 4-byte input argument, which is passed through the stack, and



```
Prepare:
   pushl  0x4(%esp)
   popl  ExtensionStack
   movl  %esp, SP2
   movl  %ebp, BP2
   push  ExtensionStackSegment
   pushl  ExtensionStackPointer
   push  ExtensionCodeSegment
   push  Transfer
   lret

AppCallGate:
   mov SP2, %esp
   mov BP2, %ebp
   ret

Transfer:
   call  ExtensionFunction
   lcall  AppCallGateNum
```

**Figure 6**. Calling an extension goes through `Prepare` and `Transfer`, whereas the return path goes through `Transfer` and `AppCallGate`. `Prepare`'s first two instructions copy the extension call's input argument to the extension's stack. The next four instructions save the stack and base pointers of the application segment so that later on `AppCallGate` can restore them. Finally the four instructions above `lret` synthesizes an artificial activation record in the extension stack for `lret`.

return one 4-byte result, which is passed through the register file. More complicated data structures are stored in the shared data area, and input and result arguments are pointers to them.

### *4.5.2 Other kernel modifications*

In addition to the new system calls described above, *Palladium* also requires several kernel modifications. First, to ensure that an extensible application process's writable pages are always marked as PPL 0, `mmap` is modified to mark all the pages in a memory region to be mapped as PPL 0 if the memory region is writable and the process that invokes the `mmap` is at SPL 2. The actual marking is performed at the page fault time. Similarly, `mprotect` is changed to prevent an SPL 3 extension from tampering with the PPL of a memory segment at SPL 2.

In *Palladium*, the standard page fault handler needs to check whether an extension attempts to access the extended application's memory that is outside the extension segment. This check is based on the application's SPL, the SPL of the code segment of the routine that causes the page fault, and the page's PPL and permission bits. If this check fails, a `SIGSEGV` fault is delivered to the corresponding user process.

The segment/page privilege levels of a process are inherited across `fork` calls along with the entire memory map. This allows an extensible application that is already at SPL 2 to fork a copy of itself. The forked clone continues to execute at SPL 2 and inherit all the loaded extensions. On the other hand, the segment/page privilege levels of a process are not inherited across `exec` calls, because new processes by default should start at SPL 3 and only move to SPL 2 when they plan to load untrusted extensions.

To prevent extension routines at SPL 3 from making arbitrary system calls, *Palladium* first adds a new field, `taskSPL`, to each process's `task_struct` as its logical SPL. When a process starts up, its `taskSPL` is 3 until it promotes itself through `init_PL`, at which point `taskPL` is 2. Whenever the kernel receives a system call, the kernel rejects the call if calling process's `taskPL` is 2 *and* the return code segment's SPL is 3. Note that for those applications that do not call `init_PL` at start-up, the above check would fail because these applications' `taskPL` is 3. Therefore, non-*Palladium* applications still can make system calls as usual. We chose to run non-*Palladium* applications at SPL 3, as in standard Linux, to avoid disruptions to the large number of existing Linux applications. *Palladium* applications can allow their user extensions to make a selective subset of system calls by encapsulating them as application services.

To prevent "infinite loop" bugs in extension routines, *Palladium* sets a time limit on the maximal amount of CPU time that a user/kernel extension module can get in each invocation. This limit is a system parameter set by the system administrator and is enforced through explicit checks at timer interrupts. When the timer expires or when a protection error is detected, the kernel aborts the offending kernel extension and, in the case of user extensions, sends a signal to the extensible application, which is supposed to have a signal handler to deal with such errors. The current *Pal-*

| Component | Inter | Intra | Hardware |
|---|---|---|---|
| **Setting up stack** | 26 | 2 | 5 |
| **Calling function** | 34 | 3 | 22 |
| **Returning to caller** | 75 | 3 | 44 |
| **Restoring state** | 7 | 2 | 5 |
| **Total Cost** | 142 | 10 | 89 |

**Table 1**. Comparison between the invocation costs for function calls within the same protection domain (Intra), across protection domains, with (Inter) and without software overhead (Hardware). All measurements are in terms of numbers of CPU cycles from the Pentium counter.

*ladium* prototype does not perform any clean-up for aborted kernel extensions, beyond reclaiming the system resources previously allocated to these extensions.

## 5 Performance evaluation

We have built two applications on the *Palladium* prototype, one based on the user-level extension mechanism, and the other based on the kernel extension mechanism, to evaluate *Palladium*'s performance at the application level. In this section, we report the performance results from micro-benchmark and applications measurements.

### 5.1 Micro-benchmarking results

Because *Palladium*'s protection mechanism is based on hardware checks, its performance overhead consists of the cost of invoking an extension function and the one-time extension module loading time. To measure the protected function call overhead, we wrote a null function with an empty body and compiled it with `gcc 2.7.2.3` into a shared library. The code generated for this function contains only the function prologue and epilogue code. Using the Pentium counter, we measured the number of CPU cycle required to invoke this null function call using *Palladium*'s user-level extension mechanism on a Pentium 200MHz machine. The results are shown in Table 1.

The number of CPU cycles required for a protected or inter-domain procedure call in *Palladium* is 142, or 0.71 $\mu$sec for a 200MHz machine. The **Setting up stack** row is the time required to create a faked activation record and save registers. The **Calling function** row shows the time required to do the actual control transfer to the extension function. This step involves a `lret` and a `call` instruction in *Palladium*. The **Return to caller** row is the time needed to return control to the caller. This is essentially an `lcall` instruction in *Palladium*. The **Restoring state** row shows the time to restore the application to the state before it calls the extension. For unprotected function calls, this corresponds to popping registers off the stack. For *Palladium*, this involves popping registers and executing an additional `ret` instruction. Over half of *Palladium*'s inter-domain procedure call overhead is due to the time taken to return control to the application from an extension because switching the privilege level from SPL 3 to SPL 2 requires additional checks.

| Size of string (Bytes) | Unprotected call | Palladium call | Linux RPC |
|---|---|---|---|
| **32** | 2.20 | 2.79 | 349.19 |
| **64** | 4.06 | 4.65 | 352.55 |
| **128** | 7.78 | 8.37 | 374.20 |
| **256** | 15.22 | 15.97 | 423.33 |

**Table 2**. Comparison between unprotected function call, protected extension function call, and Linux RPC. All measurements are in microseconds. Each data point is an average of the results from 100 runs, with a standard deviation of less than 2% of the mean in all cases.

This one instruction, `lcall`, takes about 75 cycles. Table 1 also shows the theoretical cycle counts required for the instruction sequences used in *Palladium*'s control transfer, according to the Pentium architecture manual. The difference between the measured and theoretical cycle counts is mainly due to data/control pipeline hazards.

To the best of our knowledge, the fastest IPC mechanism on Pentium machines is reported on L4 micro-kernel [16]. L4 takes 242 cycles on a Pentium 166MHz machine for an request-reply IPC *in the best case*. This cycle count assumes that all the parameters can be passed via registers. In L4, processes share page tables as much as possible. Hence an IPC does not require a page table switch. Still, a request-reply IPC in L4 involves four protection-domain crossings, whereas *Palladium* takes only two. As a result, *Palladium* as measured on the Linux kernel is faster than the best case of L4 by 100 cycles.

To evaluate *Palladium*'s performance in a more realistic context, we wrote an artificial extension function that accepts a pointer to a string and reverses the string. We compiled this function as an extension shared library, as an unprotected function call within the address space, and as a client-server program with client and server running on the same machine using Linux's Remote Procedure Call (RPC) facility, which is socket-based and is not optimized for intra-machine RPC. We then measured the time it takes from calling such a function until control is returned to the calling program, with the size of the string varied from 32 to 256 bytes in powers of two. The results are shown in Table 2 and expressed in microseconds. Each data point in the table is an average of the results of 100 runs, with a standard deviation of less than 2% of the mean in all cases. During the experiments, the CPU cache is fully warmed up.

The Linux-RPC version is more than two orders of magnitude slower than the protected and unprotected function call versions when the input size is 32 bytes. When the data size increases to 256 bytes, the RPC version is still about 14 times slower than the protected and unprotected function call versions. This shows that the constant overhead associated with IPC is quite significant. The performance difference between a unprotected procedure call and a *Palladium*'s protected remains largely constant, about 118 cycles for the string size between 32 bytes and 128 bytes. The difference increases to 153 cycles when the string size is 256 bytes. We believe this discrepancy is due to factors unrelated to *Palladium*, because some of the 256-byte runs do show a difference of 118 cycles. Because the total processing time of this extension increases with the data size, the constant extension invocation overhead becomes less and less significant in relative terms.

*Palladium* incurs a slightly higher overhead when loading an extension module: `dlopen` and `seg_dlopen` take 400 $\mu$sec and 420 $\mu$sec, respectively. The additional step that `seg_dlopen` performs compared to `dlopen` is setting the PPL of those pages that the extended application exposes to 1. PPL marking has a start-up cost of 3000 to 5000 cycles, plus 45 cycles per page marked. That is, marking 10 pages takes 3450 to 5450 cycles, or 17.25 to 27.25 $\mu$sec, which is completely overshadowed by the dynamic library open cost.

Because *Palladium* allocates separate segments for kernel extensions, cross-segment memory references incur an additional overhead for loading the segment register, which is 2 to 3 cycles according to Intel's architecture manual, but is consistently 12 cycles from our own measurement. Since *Palladium* supports shared data areas between the kernel and kernel extensions, we expect the frequency of cross-segment data references in typical kernel extensions to be low. Note that cross-segment references are not necessary in the case of user-level extensions, because extended applications and their extension segments span the same address space range.

## 5.2 Measurements from extensible applications

We have built on top of the Apache Web server a fast Common Gateway Interface (CGI) invocation mechanism called *LibCGI* [28], which allows a CGI script written in C to be invoked as a function call rather than as a separate process as in standard CGI implementation. FastCGI [9] attempts to reduce the invocation overhead of CGI scripts by re-using existing CGI processes, thus eliminating the costs associated with `fork` and `exec`. *Palladium*'s user-level extension mechanism provides the necessary protection for the Web server from *LibCGI* scripts.

We measured the number of CGI requests that a Web server can support per second from a conventional CGI script, a FastCGI script, a LibCGI script, a protected LibCGI script using *Palladium*, and a static HTML file. Fetching a static HTML file does not involve the CGI, and thus its performance serves as the best-case reference point. Performance measurements on ApacheBench benchmark [2] were taken from a modified Apache Web Server running on a Pentium 200MHz machine with 64 MBytes of SDRAM and 2 GBytes of Disk Space. In each run, a total of 1000 requests were sent to the Web server with up to 30 requests being serviced concurrently. The Web server and its clients are connected via a 100 Mbps Fast Ethernet link, which is quiescent in all runs. Each request involves an access to a fixed HTML file that is memory-resident. The Web server can service each request directly by opening the static HTML file, reading it into memory, and writing it back to the requesting client, or by invoking a CGI script that does exactly the same thing using different CGI execution models.

Table 3 shows the throughputs when the requests are serviced by the Web Server directly and by CGI scripts under

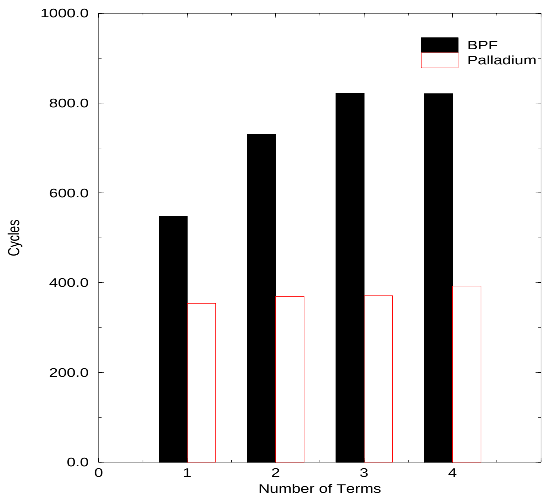


**Figure 7**. Performance comparison of a compiled filter extension and the interpreted BPF filter for a filter with a varying number of terms linked by a conjunction, when all terms are true. The measurements are in CPU cycles.

different execution models. The **Web Server** column establishes a bound on the CGI script execution throughput, since there is no CGI script invocation overhead in this case. For all data sizes, unprotected LibCGI and protected LibCGI are within 3% and 5% of the bound, respectively. This shows that LibCGI is effective in reducing the overhead of invoking CGI scripts. On the other hand, protected LibCGI is at least twice as fast as FastCGI for data size smaller than 10KBytes. A more interesting comparison is that between protected and unprotected LibCGI. The throughput of protected LibCGI is about 97.5% of that of unprotected LibCGI when the data size is 28 bytes. In all cases, protected LibCGI performs within 4% of unprotected LibCGI. This performance result demonstrates that the additional overhead that *Palladium* incurs is minimal.

When a user-level extension attempts to access pages outside its domain, such an access generates a page fault, and a SIGSEGV signal is delivered to the extended application. The latency from detecting an offending access to completing the delivery of the associated SIGSEGV signal to the application takes 3,325 cycles on the average with a standard deviation of 0.3%. In the case of kernel extensions, an offending access would cause a general protection exception, because the extension is attempting to access data beyond its segment limit. The average cost of processing such an exception is 1,020 cycles (0.5% standard deviation), excluding the extension-specific overhead associated with systems resource de-allocation. While these costs are relatively high, they are present only for misbehaving extensions and thus would not affect the critical path delay in the common case.

To evaluate the effectiveness of *Palladium*'s safe kernel extension mechanism, we built a compiled packet filter [22] that allows a filtering program written in C to be loaded into the kernel as an extension, and we measured the time to execute a packet filter rule consisting of a conjunction of multiple terms. We compared these measurements to the times required by the standard *bpf_filter* function used in TCPdump. The BPF filter essentially compiles the filter expression into

| Size of HTML file requested | Throughput (requests/sec) | | | | |
|---|---|---|---|---|---|
| | CGI | FastCGI | LibCGI (Protected) | LibCGI (Unprotected) | Web Server |
| *28 Bytes* | 98 | 193 | 437 | 448 | 460 |
| *1 KBytes* | 92 | 188 | 423 | 431 | 436 |
| *10 KBytes* | 76 | 130 | 311 | 312 | 315 |
| *100 KBytes* | 33 | 52 | 57 | 57 | 57 |

**Table 3**. Comparison of CGI, FastCGI and LibCGI in their execution throughput as measured by numbers of scripts completed per second. The client and server are connected with a 100 Mbps Ethernet link.

its own machine language and interprets the resulting expressions to evaluate the conditions. The performance comparisons are shown in Figure 7. Beyond a fixed invocation overhead, the performance overhead of the kernel-extension-based packet filter increases with a very small slope, staying almost constant. On the other hand, BPF's interpretation overhead increases significantly with the number of terms in the test packet filter rule. When the number of terms in the filter rule is 4, the extension-based packet filter is more than twice as fast as the interpreter-based packet filter. These measurements demonstrate the efficiency of *Palladium*'s kernel extension mechanism.

## 6 Conclusion

This paper describes the design, implementation, and evaluation of of an intra-address space protection mechanism called *Palladium*, which is based on the paging and segmentation protection hardware available in Intel X86 architecture. *Palladium* proves that safe and efficient user-level and kernel-level extensions that could be programmed in a similar way to standard library functions are feasible. In addition, *Palladium*'s protection domain switching overhead is the smallest among all known methods. Finally, to the best of our knowledge, *Palladium* is the first system that has successfully exploited the segmentation feature in Intel processors in a useful way. To demonstrate the effectiveness of *Palladium*, we have built a fast and safe CGI invocation mechanism that allows CGI routines to be invoked from a Web server as local function calls, and an efficient packet filter that allows packet filtering programs to run on native hardware safely inside the kernel.

We are currently pursuing several directions based on the *Palladium* prototype. First, we are planning to build a mobile code system based on *Palladium*. Combined with restricted OS services, *Palladium* could provide the security guarantee for mobile applets that are written in a compiled language such as C. Although binary portability poses a problem for this approach, the fact that a vast majority of desktop computers are Intel-based PC's renders this problem less an issue in practice. Second, we are leveraging our experiences in exploiting segmentation hardware for other purposes. For example, we are building a *protected memory* service that uses segmentation to prevent wild pointers or random software errors from corrupting specific physical memory regions. Third, better programming tools for extensions programming are needed, in particular, segmentation-aware debuggers and stub code generators to synthesize application or kernel services for extensions. Finally, we are building more extensible applications, especially in the areas of database systems and 3D graphics software, to gain more usage and performance experiences on *Palladium*.

## Acknowledgment

This paper has benefited significantly from the comments from SOSP reviewers, especially from several iterations of detailed reviews from our SOSP shepherd, Dr. Fred Schneider. This research is supported by an NSF Career Award MIP-9502067, NSF MIP-9710622, NSF IRI-9711635, NSF EIA-9818342, NSF ANI-9814934, a contract 95F138600000 from Community Management Staff's Massive Digital Data System Program, USENIX student research grants, as well as fundings from Sandia National Laboratory, Reuters Information Technology Inc., and Computer Associates/Cheyenne Inc.

## References

[1] Alexander, D.S.; Arbaugh, W.A.; Keromytis, A.D.; Smith, J.M., "A secure active network environment architecture: realization in SwitchWare," *IEEE Network*, **12**(3):37-45, May-June 1998.

[2] Apache Server project, http://www.apache.org/

[3] Banerji, A.; Tracey, J.M.; Cohn, D.L., "Protected shared libraries-a new approach to modularity and sharing," *Proceedings of the USENIX 1997 Annual Technical Conference*, p. 59-75, Anaheim, CA, Jan. 1997.

[4] Bensoussan, A.; Clingen, C.T.; Daley, R.C., "The Multics virtual memory: concepts and design," *Communications of the ACM*, **15**(5):308-18, May 1972.

[5] Bershad, B.N.; Anderson, T.E.; Lazowska, E.D., "Lightweight remote procedure call", *ACM Transactions on Computer Systems*, **8**(1):37-55, Feb. 1990.

[6] Bershad, B.N.; Savage, S.; Pardyak, P.; Sirer, E.G.; Fiuczynski, M.E.; Becker, D.; Chambers, C.; Eggers, S., "Extensibility, safety and performance in the SPIN operating system ," *ACM Operating Systems Review*, **29**(5):267-84, Dec. 1995.

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

[8] Chiueh, T.; Sankaran, H.; Neogi, A., “Spout: A Distributed Engine for Safe Execution of Java Applets,” ECSL-TR-59, Experimental Computer Systems Lab., Computer Science Department, SUNY at Stony Brook, June 1999.

[9] The FastCGI Homepage, http://fastcgi. idle.com/.

[10] Gosling, J.; McGilton, H., “The Java Language Environment”, http://java.sun.com/docs/white/index.html, May 1996.

[11] Green, P., “Multics Virtual Memory - Tutorial and Reflections,” ftp://ftp.stratus.com/pub/vos/multics/pg/mvm.html.

[12] Heid, J., “Mastering Adobe Premiere 5,” *Macworld*, **16**(1):115-17, Jan. 1999.

[13] Heiser, G.; Elphinstone, K.; Vochteloo, J.; Russell, S.; Liedtke, J., “The Mungi Single-Address-Space Operating System,” *Software - Practice and Experience*, **28**(9):901-28, July 1998.

[14] *Intel Pentium Processor Family Developer’s Manual*, Volume 3: Architecture and Programming Manual, Intel Corporation, Santa Clara, CA, 1995.

[15] Kaashoek, M.F.; Engler, D.R.; Ganger, G.R.; Briceno, H.M.; Hunt, R.; Mazieres, D.; Pinckney, T.; Grimm, R.; Jannotti, J.; Mackenzie, K., “Application performance and flexibility on exokernel systems,” *ACM Operating Systems Review*, **31**(5):52-65, Dec. 1997.

[16] Liedtke, J.; Elphinstone, K.; Schonberg, S.; Hartig, H.; Heiser, G.; Islam, N.; Jaeger, T., “Achieved IPC performance (still the foundation for extensibility),” *Proceedings of the 6-th Workshop on Hot Topics in Operating Systems* (HotOS - VI), p. 28-31, May 1997.

[17] McCanne, S.; Jacobson, V., “The BSD packet filter: a new architecture for user-level packet capture,” *Proceedings of the Winter 1993 USENIX Conference*, p. 259-69, Jan. 1993.

[18] Mendelsohn, N., “Operating systems for component software environments,” *Proceedings of the 6-th Workshop on Hot Topics in Operating Systems* (HotOS-VI), p. 49-54, Cape Cod, MA., May 1997.

[19] Necula, G.C.; Lee, P., “Safe kernel extensions without run-time checking,” *ACM Operating Systems Review*, **30**(special issue):229-43, Oct. 1996.

[20] Olson, M. A., “DataBlade extensions for the Informix-Universal server,” *Proceedings IEEE COMPCON 97*, p. 143-8, Feb. 1997.

[21] *PA-RISC 2.0 Architecture Reference Manual*, Hewlett Packard Corporation, Palo Alto, CA, 1994.

[22] Pradhan, P.; Chiueh, T.; “Operating System Support for Cluster-Based Routers,” *Proceedings of the 7-th Workshop on Hot Topics in Operating Systems* (HotOS - VII), p. 76-81, Rio Rico, AZ, March 1999.

[23] Seltzer, M.I.; Endo, Y.; Small, C.; Smith, K.A., “Dealing with disaster: surviving misbehaved kernel extensions,” *ACM Operating Systems Review*, **30**(special issue):213-27, Oct. 1996.

[24] Small, C.; Seltzer, M.I.; “A comparison of OS extension technologies,” *Proceedings of the USENIX 1996 Annual Technical Conference*, p. 41-54, San Diego, CA, Jan. 1996.

[25] Small, C.; Seltzer, M., “MiSFIT: constructing safe extensible systems,” *IEEE Concurrency*, **6**(3):34-41, July-Sept. 1998.

[26] Stonebraker, M.; Kemnitz, G., “The POSTGRES next-generation database management system ,” *Communications of the ACM*, **34**(10):78-92, Oct. 1991.

[27] Tennenhouse, D.L.; Smith, J.M.; Sincoskie, W.D.; Wetherall, D.J.; Minden, G.J., “A survey of active network research,” *IEEE Communications Magazine*, **35**(1):80-6, Jan. 1997.

[28] Venkitachalam, G.; Chiueh, T.; “High Performance Common Gateway Interface Invocation,” *Proceedings of 1999 IEEE Workshop on Internet Applications* (WIAPP ’99), p. 4-11, San Jose, CA, July 1999.

[29] Wahbe, R.; Lucco, S.; Anderson, T.E.; Graham, S.L., “Efficient software-based fault isolation,” *ACM Operating Systems Review*, **27**(5):203-16, Dec. 1993.

[30] Wilkes, J.; Fouts, M.; Corrors, T.; Hoyle, S.; Sears, B.; Sullivan, T., “Brevix design 1.01,” HPL-OSR-93-22, Hewlett-Packard Laboratories, Palo Alto, Apr. 1993.